DISK ENCLOSED

ON DISK: HotLinks Developers Kit from SoftLogik

AC's TECH / AMIGA

For The Commodore

Volume 3 Number 1 US $14.95 Canada $19.95

Linking with HotLinks

- Comeau Computing's C++ Reviewed
- ARexx System Log
- Programming the Amiga in Assembly Language, Part 5—Dinner with Julia
- True BASIC Extensions
- Bitmapped Graphics
- Transformer
- Trading Commodities in Workbench 2.0
- Using the Audio Device
- Make Your Own 3-D Vegetation

01

Contents

Volume 3, Number 1 AC's TECH AMIGA

Departments

```
printf("Hello");

print "Hello"

JSR printMsg

say "Hello"

writeln("Hello")
```

Whatever language you speak, AC's TECH provides a platform for both gaining insight and sharing information on its most innovative implementation for the Amiga. Why not see if your latest programming endeavor can help a fellow Amiga user expand upon his or her vocabulary? To be considered for publication in AC's TECH, submit your technically oriented article (both hard copy & disk) to:

AC's TECH Submissions
PiM Publications, Inc.
P.O. Box 2140
Fall River, MA 02722-2140

AC's TECH AMIGA

ADMINISTRATION

Publisher:	Joyce Hicks
Assistant Publisher:	Robert J. Hicks
Administrative Asst.:	Donna Viveiros
Circulation Manager:	Doris Gamble
Asst. Circulation:	Traci Desmarais
Traffic Manager:	Robert Gamble
Marketing Manager:	Ernest P. Viveiros Sr.

EDITORIAL

Managing Editor:	Don Hicks
Editor:	Jeffrey Gamble
Hardware Editor:	Ernest P. Viveiros Sr.
Senior Copy Editor:	Paul Larrivee
Copy Editor:	Elizabeth Harris
Video Consultant:	Frank McMahon
Art Director:	Richard Hess
Illustrator:	Brian Fox
Editorial Assistant:	Torrey Adams

ADVERTISING SALES
Advertising Manager: Wayne Arruda

1-508-678-4200
1-800-345-3360
FAX 1-508-675-6002

AC's TECH For The Commodore Amiga™ (ISSN 1053-7929) is published quarterly by PiM Publications, Inc. One Currant Road, P.O. Box 2140, Fall River, MA 02722-2140.

Subscriptions in the U.S., 4 issues for $44.95; in Canada & Mexico surface, $52.95; foreign surface for $56.95.

Application to mail at Second-Class postage rates pending at Fall River, MA 02722.

POSTMASTER: Send address changes to PiM Publications Inc., P.O. Box 2140, Fall River, MA 02722-2140. Printed in the U.S.A.

First Class or Air Mail rates available upon request. PiM Publications, Inc. maintains the right to refuse any advertising.

PiM Publications Inc. is not obligated to return unsolicited materials. All requested returns must be received with a Self Addressed Stamped Mailer.

Send article submissions in both manuscript and disk format with your name, address, telephone, and Social Security Number on each to the Editor. Requests for Author's Guides should be directed to the address listed above.

Startup-Sequence

Small Openings Can Yield Big Opportunities

At the Fall '92 COMDEX in Las Vegas, Commodore Business Machines launched the third new Amiga into the U. S. marketplace in as many months. The newest Amiga, the A1200, appears to be a cross between the other two new additions to the Amiga line, the A4000 and the A600.

Like the Amiga 4000, the A1200 uses the Advanced Graphics Chip Set as well as AmigaDOS 3.0. With a price of $699 for an A1200 less hard drive, the Amiga user now has a less expensive option for getting into the new AA Amiga graphics chips. (For more on the A1200, please see the January issue of *Amazing Computing*.)

PCMCIA

Products such as the 40MB hard drives (above) by Integral Peripherals and the 2 MB expansion RAM for the A600 and A1200 (left) by New Media Corporation offer more than something to fill a slot.

The Amiga 1200 looks remarkably like an updated Amiga 500—or possibly a stretch version of the Amiga 600—with an off-white case and a memory board expansion panel access on the underside. It also has an additional similarity to the Amiga 600 that has not always been well received by the Amiga developer community. In lieu of the access slot to the Amiga BUS, both the A600 and the Amiga 1200 have adopted the PCMCIA standard interface for expansion devices.

When this feature was introduced with the Amiga 600, many Amiga developers were greatly upset and with good reason. They had spent time, money, and other resources to develop peripherals based on the standard access slot found in some configuration in every Amiga since the original introduction of the Amiga 1000 in 1985. The slot has seen some very exciting expansion products, especially for the Amiga 500.

A500 memory expansions, hard drives, IBM emulation, and more have taken advantage of the intimate, base-level connection this port provides. However, the PCMCIA port available on the Amiga 600 and the Amiga 1200, was created for a larger audience.

The PCMCIA standard type 2 slot on the A600 and A1200 was developed as an expansion standard for the PC and laptop community. There was a need to supply one standard that a variety of highly specialized devices could use. The result of this standard has been everything from RAM and bubble memory to complex devices such as FAX/modems and 40MB hard drives in devices no larger than a credit card.

By making this standard a port on the Amiga 1200 and 600, Commodore has not only opened the door to its users for a wide variety of hardware capabilities, it has also opened a door to its developers to provide a wider audience for their products.

What Is PCMCIA?

PCMCIA is an acronym for the Personal Computer Memory Card International Association, a standards organization with over 300 member companies. PCMCIA has created the standards for the Personal Computer Card or PC Card which is often referred to by the standard's own name of PCMCIA. Originally developed for the portable market, the standards were designed to provide a high-level environment for new products. Now the PCMCIA standard is being used by desktop computers such as IBM and, of course, Commodore.

The PCMCIA standards cover the physical dimensions, electrical specifications, and the software architecture for PC Cards. There are three physical size specifications in the PCMCIA standards: Type I, Type II, and Type III. While all three specifications maintain the same 68-pin connector and the same length and width dimensions, they do differ in thickness. Type I cards are 3.3 mm thick, Type II cards are 5.0 mm thick, and Type III cards are 10.5 mm thick. These variances allow a standard to apply to a small 40MB hard drive as well as a 2MB RAM memory expansion.

Under the standard, it is possible to construct a wide variety of computer peripherals and devices. As other hardware manufacturers follow the lead of IBM, Commodore, and a variety of laptop computer makers, there will be a larger audience of potential customers. This combination of a variety of products to an expanding audience provides Amiga hardware manufacturers with possibilities they have never had before.

Learning More

In later issues we will follow the work that Commodore is doing with the PCMCIA standard as well as keep track of what is being created in the PC arena. The PCMCIA was offering special discounts on their documentation at COMDEX, good on prepaid orders received before December 31, 1992. For more information on the standard itself and any other help from the PCMCIA organization, please contact:

PCMCIA
1030G East Duane Avenue
Sunnyvale, CA 94086
telephone (408) 720-0107
FAX (408) 720-9416
BBS (408) 720-9388

Who Needs the Market?

While I am sure there are Amiga developers who feel there is little to be gained by attacking this potential market, it is important to note that new developers are providing products for the Amiga 600 and Amiga 1200 with PCMCIA interfaces. The latest release was announced at COMDEX by New Media Corporation.

New Media has developed a card with up to 4MB of additional RAM through the PCMCIA port. At $240 for the 2MB PSRAM card and $385 for the 4MB PSRAM card, New Media could find a very profitable market for this much-needed expansion. For more information on New Media and their card, contact Mr. Simon Harvey, V. P. of Sales & Marketing, New Media Corporation, Irvine Spectrum, 15375 Barranca Parkway, Building B, Irvine CA 92718.

Opportunity?

One interesting facet of the Amiga is that it has always remained a trend setter in ideas and potential markets. PCMCIA is a portal by which Amiga developers can not only provide product for the Amiga, but they can look at the demands, needs, and specifications of other markets. Opportunity does not always knock; sometimes it is accessible through a very small opening.

Sincerely,

Don Hicks
Managing Editor

C++

You've never heard of Comeau Computing? Neither had I until I saw an advertisement for *Comeau C++* in *AC's Guide To The Commodore Amiga*. According to the press release I received with Comeau C++, Comeau Computing was formed in 1985 as a privately held consulting firm specializing in UNIX and C. The company develops and sells programming tools for a variety of hardware platforms. Its primary commercial products appear to be Comeau C++ and a shell program named CCsh. Comeau C++ is available for many UNIX systems (including various flavors of UNIX), MS-DOS systems, and for AmigaDOS. The company is a founding and voting member of the ANSI C++ Committee.

C++ is a general-purpose programming language based on C. It was developed in the early 1980's at AT&T's Bell Laboratories and became publicly available in 1985. Since then, the language has undergone several revisions. The current revision is 3.0, which is defined in the version of the C++ reference manual published in February 1990. This version of the reference manual is the base document for ANSI standardization of the language. In March 1992, revision 3.0 of C++ became available for the Amiga when Comeau Computing began shipping an AmigaDOS version of their C++ development system called Comeau C++ 3.0 With Templates.

COMEAU C++

The heart of Comeau C++ is a licensed port of cfront ("C front-end") from AT&T's Unix System Laboratory (USL). This is the de facto standard C++ compiler. It is the central part of USL's C++ Language System. Technically, cfront is a compiler, but most programmers refer to it as a translator since it translates C++ source code into C source code. The generated C source code is then compiled by any standard C compiler ("C back-end") to produce native code for a specific machine. Because of this, Comeau C++ is not a stand-alone C++ development system. You also need a C compiler and standard C link libraries to create executable C++ programs with Comeau C++. The C compilers that can be used with the AmigaDOS version of Comeau C++ are SAS C and Aztec C. The documentation states that Matt Dillon's C compiler system, DICE, may also work with Comeau C++ in the future.

In addition to the C++ compiler, Comeau C++ includes an install script, documentation, a C preprocessor, a C++ link library, standard C++ include files, and an AREXX script for compiling and linking C++ programs. The software is on two disks and the documentation is in a standard vinyl three-ring binder. The total cost of the package is $250.00. The price includes shipping and handling. It also includes free lifetime technical support.

INSTALLING COMEAU C++

Installing Comeau C++ is a snap: you simply put Disk 1 in drive 0, change to the DF0: directory, and type "rx install". The rest is automatic. You are prompted once for the name of the directory where your C include files are kept, and once to change disks. The entire installation process requires less than five minutes on an Amiga 3000. The only part of the installation process that is not completely automatic is adding various logical directory assignments to your startup file. A file named forS containing logical directory names for the Comeau C++ directories is provided with the system. You can copy this file into your startup script. After you finish installing Comeau C++, you verify that everything works correctly by compiling C and C++ versions of the famous "Hello, world" program and running them.

The installation process creates a directory named Ccomeau. All the components of the system are in this directory. It includes several sub-directories, two "Hello, world" test programs, and some documentation. The sub-directories of Ccomeau are ARexx30, c30, include30, and lib30. Before discussing the contents of these directories, I need to describe C++ classes and objects, and describe how objects are created and destroyed in C++.

C++ CLASSES, OBJECTS, CONSTRUCTORS, AND DESTRUCTORS

The following is a C++ class definition:

```
class String
{
        char *_str;         // private member
public:
        String( char *str ) // String constructor
        {
                _str = new char[strlen(str)+1];
                strcpy(_str, str);
                _str[strlen(str)] = '\0';
        }
```

A Look at Comeau Computing's C++ 3.0

by Forest W. Arnold

```
        ~String()           // String destructor
        {
                delete []_str;
        }
        // pointer and reference output operators
friend ostream &operator<<(ostream&,String*);
friend ostream &operator<<(ostream&,String&);
};
```

This class definition defines a type named "String". Classes consist of objects together with the operations that can be applied to the classes' objects. Objects are a class's variables. As defined above, String objects consist of a single data element: a pointer to a dynamically allocated character array. The operations that can be applied to String objects are String creation, String destruction, and output of a String's value. The procedure String() is called a constructor, and the procedure ~String() is called a destructor. Constructors initialize newly created objects, and destructors clean them up before the memory they occupy is made available for reuse. The C++ compiler is responsible for making sure that a class's constructors are called when its objects are created, and making sure that its destructors are called when its objects are destroyed. Objects are created by declaring them, or by dynamically allocating them using the C++ operator new(). They are destroyed when the scope within which they are declared is exited or with the C++ operator delete(). Listing 1 is a simple but complete C++ program that creates two String objects, prints their values using an overloaded C++ output operator (<<), and deletes the dynamically created String object. Listing 1 also presents the output produced by this program. You will notice from the program's output that both the String constructor and destructor are called for each object, even though one is created by declaring it outside any procedure (making its scope "global") and it is never explicitly destroyed. I will explain below how the C++ compiler makes sure constructors and destructors for statically allocated global objects are called.

COMPONENTS OF COMEAU C++

I mentioned above that all the software for Comeau C++ is contained in several sub-directories of a directory named Comeau. The following paragraphs describe the contents of each of these sub-directories.

The ARexx30 directory contains only a single file, como.rexx. This is an ARexx script that creates an executable program from a C++ source code file. After you create the source code file for your C++ program, you compile and link it using como.rexx. The command to compile and link the String program in Listing 1 is "como -ostring string.c". This script works well for small programs that only need linking with a few libraries. If your programs require include files, link libraries, and object files from several different directories, you will need to modify the script, or use a "make" file to compile and link them. Listings 2 through 5 are sample make and link "with" files for compiling and linking C++ programs using Comeau C++ and SAS C. The make file in Listing 2 and the "with" files in Listings 3 and 4 compile and link a C++ program, and the make file in Listing 5 compiles and adds C++ modules to a link library.

The c30 directory contains one utility program, named include, and the three programs you use to compile C++ source files. These programs are dcpp, cfront, and std. dcpp is a C preprocessor written by Matt Dillon. C preprocessors prepare C or C++ source files for compilers. Preprocessing involves removing comments, expanding macros, merging external files into the source file, and performing various other functions. Any C preprocessor that is ANSI compatible and recognizes BCPL-style comments ("//") works with Comeau C++, but the recommended preprocessor is dcpp. It is fast, reliable, and accurate.

cfront is the actual C++ compiler. Comeau's cfront is a licensed port of AT&T's cfront. The Comeau port supports C++ templates, a feature of C++ documented in the reference manual as experimental. Templates are "super-macros" that cfront uses to automatically generate parameterized classes and functions. Using templates, you can create "families" of classes or functions by defining a single template class or function that accepts a type as a parameter. You specify the actual type for the intended class or function as a parameter in a template declaration. cfront automatically generates code for a class or function of the specified type from the template. Templates are somewhat similar to standard C macros, but they are much more powerful.

The third program you use to create a C++ program with Comeau C++ is std. Versions of this program are provided for use with both the SAS and Aztec compilers. The Aztec version is named

astid. I think the name of the program is an abbreviation of "static initialization and destruction." The documentation for Comeau C++ does not discuss stid in any detail. It is the part of the compiler package that installs procedure calls to initialize and destroy static global objects. It generates a C source code file containing a procedure named _ConStrucT(). The C++ compiler places a call to this procedure in main() so that it is called as soon as a C++ program begins executing. The compiler also generates functions that call constructors and destructors for static global objects. _ConStrucT() calls the initialization procedures when a C++ program begins execution, and arranges for the cleanup procedures to be called when the program exits. How does stid know the name of the procedures that cfront generates so they can be called by _ConStrucT()? It finds the names by examining the program's link map. The names of the procedures cfront generates to call the constructors and destructors for static global objects are prefixed with "_sti_" and "_std_", respectively. stid looks at the symbols in the program's link map and generates calls to the procedures it finds whose names contain these prefixes. Because stid needs to examine a program's link map, creating a C++ program with Comeau C++ requires two link steps. The first link step creates a link map of the program's symbols. After this step finishes, stid executes. It creates a new source code file containing _ConStrucT(). This source file is then compiled (using your C compiler) and the final link step is run to put everything together into an executable program.

The utility program in this directory, include, is used during the installation process to create the Comeau C++ include directory. It can also be used in a stand-alone mode. The C++ compiler encodes all names it finds in a source file. It does this to support overloaded function names (the same name is used for different functions) and to implement type-safe linkage. Most programmers refer to C++'s name encoding scheme as "name mangling." There are two problems that result from name-encoding. First, if references to symbol names in existing libraries are mangled, the linker will be unable to locate them and link the program. Second, if you are debugging a C++ program with a source-level debugger, the mangled names recognized by the debugger will not correspond to the unmangled names in the source file. The first problem is easily solved. You prevent cfront from mangling the names of existing C variable and procedure names by declaring them as follows:

```
extern "C" {
names of C variables and functions
}
```

The simplest way to protect existing C variable and function names from name encoding is to protect the entire header file in which they are declared:

```
extern "C" {
#include "existing_include_file"
}
```

This is exactly what the include program does. It creates a C++ include directory that parallels your existing C include directory. Each C++ include file simply protects an existing C include file of the same name. The second problem resulting from name encoding is not as easily solved as the first problem. Its solution requires support for name unmangling at the source code debugger level. This level of support must be provided in the debugger, and SAS's CodeProbe debugger does not decode mangled C++ names. There are "filter" utility programs that unmangle encoded C++ names, and the Comeau documentation refers to an unmangler program named comofilt. However, this program is not included with the AmigaDOS version of C++.

The include30 sub-directory contains the Comeau C++ include files. These are C++ versions of the include files in your standard include directory (generated by the include program discussed above), several Comeau include files, and include files that are part of the C++ Language System from AT&T.

The lib30 sub-directory contains two link libraries. One is for SAS C and the other is for Aztec C. Both libraries contain ports of the stream and io manipulator procedures that are distributed with C++, plus other procedures that are part of the core distribution of C++.

COMPILING AND LINKING A C++ PROGRAM

If you are compiling and linking a simple C++ program, you can use the rexx script provided with Comeau C++. This is the easiest way to use Comeau C++. If you want to create your own C++ link libraries or link a complex program, you need to know how all the programs described above work with each other and with your C compiler and linker. Following is a step-by-step description of how a C++ executable program is generated (steps are for SAS C, but the steps for Aztec C are similar).

1. Create a C++ source code file.

2. Preprocess the source file. The preprocessor recommended for use with Comeau C++ is cpp. The generated file name has a suffix of ".p".

3. Translate the preprocessed C++ source file to create a temporary C source file. This is done using cfront. The name of the generated file has a suffix of ".c". You should not place this file in the same directory with your original source code file, unless you modify the name. I place the file in the T: directory, and add a "T" to the end of the file name.

4. Compile the temporary source file generated in step 3 using your C compiler.

5. Generate a symbol table for the program with a linker (blink or slink if you are using SAS C). The default name for the map file containing the program's symbols is CCprelink.map. I use the name of the source file with a ".map" suffix for the name of the map file.

6. Run stid (or astid). Input to this program is the map file generated in step 5. This step creates a source code file to take care of static initialization and destruction. The file name is the same as the name of the original source code file, but the name is prefixed with an underscore.

7. Compile the file generated in step 6.

8. Run a final link. This step links all the object files generated in the above steps and any required library procedures to produce an executable C++ program.

Listings 2 through 4 contain a make file and "with" files for compiling and linking the sample String program. These files show how the steps listed above are used to compile and link a C++ program. The make and with files are for version 6 of the SAS compiler and linker. Creating a link library from C++ source files is similar to creating an executable program. The only difference is that the object modules are placed into a library instead of being linked into a program. Listing 5 is a sample make file for compiling several C++ source files and placing the object modules into a library.

DOCUMENTATION

The documentation for Comeau C++ is sparse. Most of it is a reprint of the C++ reference manual. The "User's Manual for AmigaDOS" portion of the documentation consists of an overwhelming total of eight pages. Three of these eight pages are "man" pages for running como.rexx, and two pages are installation instructions. In all fairness, the documentation is adequate for compiling and linking simple C++ programs. It also documents all the different input flags for the programs discussed above, and discusses some potential problems you may encounter if you are using version 5.10 of SAS C with Comeau C++. However, there is no documentation that tells you how to create your own C++ libraries or how to link programs with libraries other than your standard compiler libraries and the C++ libraries. You have to examine como.rexx to figure out how to compile and link a complex C++ program or create a link library. As an alternative, you can refer to this article!

SUPPORT

Free technical support for Comeau C++ is available in several different forms. You can write or call, or obtain support through BIX, Compuserve, Prodigy, or usenet. I called once to ask several questions. My call was returned within several hours. The person with whom I spoke was polite and courteous, and I was satisfied with the answers to my questions.

SYSTEM REQUIREMENTS

To use Comeau C++, you will need at least 1MB of memory. The C++ compiler, cfront, is over 0.5MB in size, and requires about 100,000 bytes of stack space. I would not recommend using C++ without 2MB of memory and a hard disk. You can conceivably use it without a hard disk, but you will probably need enough memory to load cfront and stid into RAM. You will also need enough memory to run your C compiler and linker.

HOW WELL DOES COMEAU C++ WORK, AND HOW COMPLETE IS IT?

I have used Comeau C++ for six months. I have created many small programs and C++ libraries with Comeau C++. To really test Comeau C++, I used it to compile a large, complex, UNIX-based C++ program that I transferred to my Amiga. I have experienced no problems with C++, and as far as I can determine, Comeau C++ is faithful to the specifications in the reference manual. This was expected since cfront is licensed from AT&T, but problems can (and do) creep into software during porting. I did encounter two problems using the 5.10b version of the SAS C compiler. I reported the problems to SAS, and they acknowledged that the problems were C compiler bugs. Both of the problems are fixed in Release 6.0 of the SAS C compiler. (Release 6 of SAS C is excellent!)

The Release 5 version of the CodeProbe source-level debugger does not work with Comeau C++, but the Release 6 version works just fine. To use the debugger, you must use the "Sourcels" compiler option to get the name of the original C++ source file placed into the object module. Otherwise, CodeProbe thinks the source file you are debugging is the temporary C file generated by cfront. As I mentioned above, CodeProbe does not unmangle symbol names, so you need to look at the temporary C file to determine the mangled names of the symbols you are interested in examining with the debugger.

Some symbol names in the version 6 SAS libraries are different from their version 5 names. They were changed to conform to ANSI C requirements. As a result of the name changes, if you link with libraries created with version 5 of the SAS compiler, you will get "undefined name" errors from slink. The correct solution to this problem is to obtain a "patch" to the libraries. I contacted Comeau, and was told they were sending a library patch to support version 6 of SAS C. This problem can also be corrected by redefining the old symbol names to their new names. The link files in listings 3 and 4 show how this is done for the symbols in the Comeau C++ libraries that were renamed.

Two sets of library classes and procedures that are part of the C++ Language System on most UNIX workstations are not included with Comeau C++. These are classes and procedures that implement co-routines and complex arithmetic. Co-routines implement a parallel processing algorithm using a "task" class. Co-routines execute sequentially instead of in parallel, but programs that use the co-routine procedures are easy to port to architectures that support parallel processing. The complex arithmetic classes and procedures implement complex numbers (numbers that have a real part and an imaginary part). Both of these sets of classes and procedures address specialized problem areas. These classes and procedures are not part of Comeau C++. This is not a serious omission (unless you are writing programs that require them). Otherwise, Comeau C++ seems to be a complete port of AT&T's C++ Language System.

OVERALL EVALUATION OF COMEAU C++

Here is how I rate Comeau C++:

Technical quality and reliability:	excellent
Price:	good
Support:	excellent

Installation:	excellent
Documentation:	poor
Completeness:	excellent
Integration with AmigaDOS:	excellent
Integration with C Compilers:	excellent

The above ratings are entirely subjective, and other than price, are based on comparing Comeau C++ with the professional programming tools and compilers I use on UNIX workstations. Compared with prices for C++ development systems on UNIX workstations (around $2000.00), even the price of Comeau C++ is excellent! Except for the documentation, I am entirely satisfied with Comeau C++.

FURTHER READING

If you are interested in learning more about C++, and about C++ Release 3.0 in particular, The C++ Programming Language, second edition, is an excellent and interesting source of information. Its author is Bjarne Stroustrup, the "father of C++." It is published by Addison-Wesley. The book has over 600 pages and covers all aspects of revision 3.0 of C++. It also contains the entire C++ reference manual.

Listing 1. Sample C++ Program

Listing 2. string.make

Listing 3. string.lnk1

Listing 4. string.lnk2

Listing 5. linklib.make

✓

Please write to:
Forest W. Arnold
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

SysLog:

What the heck is a an ARexx command host? It is a question often uttered by beginner ARexx programmers as they wade their way through the ARexx User's reference manual. Well an ARexx command host is usually an application program which commonly has been written in C or assembler and accepts commands via an ARexx port. The user specifies to the ARexx which command host to use with the "address" command. ARexx will then send any command found in an ARexx script which it is does not recognize on to the specified host. In this way several applications which supply ARexx ports can be "glued" together with ARexx to provide super-applications. Also, several applications now use ARexx as their macro language, where the ARexx macros that a user writes are sent to the ARexx interpreter which in turn sends commands back to the application to perform the user's wishes.

It is also possible to write simple command hosts in ARexx itself. Why would somebody want to write a command host in ARexx? As ARexx is an interperative language, just like AmigaBasic, and thus can be very slow when compared to programs written in compiled languages like C or assembler, it would seem strange to write a host in ARexx. While this argument is true, there are advantages to writing a command host in ARexx, if only just to prototype the final application. Since it is interperative, you don't have to compile your program and you save time while trying out new ideas. ARexx also has an interactive debugger, built-in functions which make it easy to write a command host. It is a great way to learn a bit more about the way in which ARexx works.

The rest of this article will show how to write a simple ARexx command host which can be used to log from other ARexx scripts to a log file. This is a similar application to the one described in Brian Zupke's article "Message Logger" in the August 1991 issue of *Amazing Computing*. In Mr. Zupke's article, however, all programs are written in C and do not use an ARexx port.

The use of a log file is very common in operating systems such as UNIX, where any errors or system messages are saved to various log files. One such file is know as the "syslog" file. A program running under the UNIX operating system can write to the log file by making calls to special library functions. AmigaDOS does not have such a facility but as Mr. Zupke's article shows, and I hope this one will as well, it is easy to add. Our ARexx command host will add messages sent to it, together with a time stamp, to a log file. You will be able to add commands to your ARexx programs and even your shell scripts to send messages to the logger host for inclusion in the log file.

You may well ask, Why don't we just write directly to the log file? Well, AmigaDOS like UNIX is a multitasking operating system, so it is possible that more than one process will try to write to the log file at any time. If this were to occur, a number of things might happen; both writes could work, one write could fail since the file could be locked by the other process or the messages will appear intermingled in the log file. So some sort of control is required to "serialize" the log requests. On Unix systems this control is supplied by a special deamon process know as "syslogd". This process accepts messages sent to it by other processes and writes them to the file one at a time. The logger host will perform a similar job for the Amiga.

Our simple host will accept, via a public port, two commands:

`LR_LOG message`

On receiving this command the logger will write the "message" together with a time stamp to the log file, "s:syslog".

`LR_STOP message`

On receiving this command a shutdown message will be output to the log file, together with a time stamp and the contents of "message".

It is very important that all packets are returned to the process which sent them.

Listing 1, logger.rexx, is a complete listing of the ARexx logging host, which you may want to type in now and try out. ARexx must already be running on your machine before you can start the logger. If ARexx is not already running, you can start it from a shell command line with

```
rexxmast >NIL:
```

To start the logger itself, use the following command from a shell window:

```
run >NIL: rx logger.rexx
```

To check that the host has started have a look in the log file "s:syslog", for a startup message. You can add messages to the log file from a shell window with the following:

```
rx "LR_LOG JUST A TEST"
```

If you now look at the log file, there should be another entry, "JUST A TEST", added. You could also use the ARexx program in Listing 2, "log.rexx", to add messages.

An ARexx Host for Logging Messages

By Paul Gittings

```
rx log Another Test
```

The above will add another entry to the log file which will say "Another Test".

The following line when added to an ARexx program will log the message "a log message" to the log file:

```
'LR_LOG' "a log message"
```

To shut down the logger, you have two choices; you can use the program in Listing 3, "stoplog.rexx", or you can enter the following in a shell window:

```
rx "'LR_STOP' Bye!"
```

If you intend to use the logger in your system you should copy the three ARexx programs to "REXX:", assigned to the directory where you keep your ARexx scripts. You should also start the logger in your "s:startup-sequence". Users of 2.0 should note that "rexxmast" is not run until after the "s:User-startup" has been executed. These users will either have to run the logger from "s:startup-sequence" (after rexxmast is run) or move the line that runs rexxmast to their "s:user-startup" file.

Now you've seen the "logger.rexx" in action, let's have a closer look at the code; you might want to read the sidebar "ARexx programming in Style" to better understand the coding conventions that I have followed in writing these scripts. The first major piece of work is to ensure that the ARexx support library "rexxsupport.library" is present, since the logger host will use several functions that this library supplies.

```
if( ~Show('Libraries', "rexxsupport.library") ) then
do
        if( ~AddLib("rexxsupport.library", 0, -30 ,0)
) then do
                call OutputLogMsg( ,
                        "SysLog: ERROR : Cannot load
RexxSupport.library")
                exit 20
        end
end
```

In the above, a check is made to see if the library is loaded. The "Show()" function will return a Boolean flag which indicates if the specified library appears in the system's list of Libraries. Since the check is for the instance where the library is not there, the return code from "Show()" is negated with the "~" operator. If the library is not present, it is loaded by a call to "AddLib". The arguments "0, -30, 0" are taken from the ARexx manual and you need not worry about them other than getting them right. If for any reason the load fails, an error message is displayed and the program exits.

Since the intention is that other processe send the host messages, the host needs to create a destination for these messages. This destination is a public port. In ARexx, creating a public port is easy; all that is required is a call to "OpenPort()" with the name of the port to be created.

```
if( ~OpenPort(OURPORTNAME) ) then do
        call OutputLogMsg( ,
                "SysLog: ERROR : Cannot create port")
        exit 20
end
```

The variable "OURPORTNAME" has the value "SYSLOG" so the port that was created is called "SYSLOG". Again the function "CreatePort()" returns a Boolean value and, as is good practice, this return value is checked. If the returned result was false, then some error has occurred and the port was not opened. One reason why a port cannot be created is that a port with the same name already exists. If the port could not be opened, the program exits.

There is one last thing to be done before the host starts processing messages and that is to log a startup message in the log file with a call to "OutputLogMsg()".

```
call OutputLogMsg("SysLog: StartUp" )
```

Before you run off and look this function up in your ARexx manual, I should tell you that this function is an internal function to the logger host and can be found in listing 1. The function starts at the label

```
OutputLogMsg:
```

I will come back to this function a little later.

The logger host has finished all its startup preparations and is now ready to process messages. Messages arrive at the host in structures called packets and until the host receives a packet there is nothing to be done, so the host waits for a packet to arrive

```
call WaitPkt( OURPORTNAME )
```

The "WaitPkt()" function will not return until a packet is available at the named port. When a packet is available, it should can be pulled off the port's message queue:

```
Packet = GetPkt( OURPORTNAME )
```

"Packet" now contains the address of (pointer to) a message structure. It turns out that on occasions "WaitPkt()" returns when there is an empty packet on the queue. These packets seem to be generated by ARexx when debugging is turned on via the program TS; here may be other occasions as well. The host should ignore empty packets! If a packet is empty, then the pointer that "GetPkt()" returns will be a special pointer, a null pointer. A null pointer is a pointer which points to nothing.

ARexx supplies a function, "Null()", which returns a null pointer. So to check if "Packet" contains a null pointer, it is compared to the value returned by the function "Null()". If "Packet" contains a null pointer, the logger goes back and waits for another packet to arrive. If "Packet" does not contain a null pointer, the command that was sent to the host is retrieved from the packet by a call to "GetArg()". This returns the command and the arguments, sent to the host, as a string. A parse instruction is used to split the string up into its constituent parts. The "parse" instruction will search for the first occurrence of a space, " ", in the string. Everything before this space is assigned to the variable "Command"; everything after the space will be assigned to "LogMsg". To make processing of the command easier, it is also converted to upper case.

```
if( Packet ~= Null() ) then do
        parse value GetArg(Packet) with Command ' '
LogMsg
        upper Command
```

It is very important that all packets are returned to the process which sent them. The process which sent the packet will not be

Side Bar: ARexx Programming with Style

I have only recently started programming in ARexx and one of the first things I did was to create an "ARexx programming style guide." Style is usually associated with clothes, cars, manners, etc., but not with programming. So what do I mean by a programming style?

A programming style is simply a set of rules, or guidelines, that you follow when writing a program. Most software departments in companies have their own "style guides" which their programmers are expected to follow when writing their code. The aim of these rules is to make the code easier for the programmer and others to read. This allows programmers to pick up code written by other programmers and for them to feel comfortable with the layout of the programs. This is very important as it is usual for new code to be reviewed in a "code walk-through" by a group of programmers, most of whom have no or little involvement in writing the code. The purpose of the review is for the reviewers to "walk" through the code a line at the time figuring out what the program is doing and looking for bugs. Even though the programs you write may not be reviewed, by following a simple set of rules you will make life easier for yourself if you ever have to return to a program to add or modify features.

Another reason for style rules is that it can reduce bugs and wasted time! A good set of style rules promotes good programming practices and avoids sloppy coding methods which introduce or hide a problem in your code.

The following is a basis for an ARexx style guide. Now this guide works for me. You may find some of the rules awkward or restrictive when coding; feel free to modify them to your needs but ensure that you have a valid reason for the changes.

COMMENTS:

The first rule is actually dictated by ARexx. Always start a program with a comment. However, many people waste this comment by just entering:

```
/* */
```

What you should put in the first comment is the name of the ARexx script, so when you are looking at a printed listing you know which script it is a print out of. A description of what the script does, a description of the command line arguments, and an example of how to run the program.

Don't be shy with your comments: the more you put in when you write your program the easier it will be to understand the program later.

If a comment is multiple lines long, start and end the comment on a separate line:

```
/*
        This is a multi-line comment
        This is the second line  */
```

This allows lines to be added and deleted easily. Also, indent the comment so that the textual part stands out; white space is just as important in programming as in writing.

able to run until the packet is returned. To return the packet, one uses the "Reply()" function. When returning the packet a "return code" can also be specified. A non-zero return code indicates that something went wrong in processing the command. Since in our simple example the sending process will not be checking for a return code, we will just return 0.

```
        call Reply(Packet, 0)
```

At this point in time the host now has a command and a message to log. Processing of the command is fairly straightforward. The "select" instruction is used to compare the command that was received with the commands that the host understands. If the command received was "LH_LOG" the host will simply log the message. If the command had been "LH_STOP", a shutdown message would have been logged, the host's port will then be closed and the program will exit. If the command is neither of these, a message is logged to to the log file saying that an unknown command was received.

```
select
        when ( Command == "LH_STOP" ) then do
                call OutputLogMsg("Syslog: Shutdown:"
LogMsg)
                ClosePort( OURPORTNAME )
                exit 0
                end
```


NAMING CONVENTIONS

Names of variables and functions should be as descriptive as possible. If a name is made up of several words some people separate the words with an underscore "_", eg this_is_a_long_variable_name. I think it is better not to use "_" and to capitalize the first letter of each word that makes up the name eg. LongVariableName. The "_" supplies no extra information to the name and requires extra typing. Using the capitalized words also helps the function and variable names to stand out.

CONSTANTS

Many languages allow you to have named constants; in C++ the following would define a constant called "PI"

```
        const double PI = 3.141
```

now when the programmer wants to use pi in a calculation he or she can just use the name "PI" rather than typing out "3.141" all the time. If it is decided that the value of pi needs to be more accurate only one line has to be changed rather than all the places in the program where pi is used. Now for the bad news. ARexx does not have named constants. To get around this, use variables as constants! To give some indication that a variable is in fact intended to be a constant value use only upper case letters in the variable name.

```
        PI = 3.141        /* constant: value of pi */
```

A word of warning. C++ will generate an error if an attempt is made to assign another value to a constant. Since we are using variables Arexx on the other hand would see nothing wrong if you assigned a new value to PI. As you can imagine if you are not careful this could lead to some nasty bugs! Another option is to create a function for each of your constants.

These functions will just return the value you would have assigned to your named constants. For example an ARexx function which returns the value of pi would look like

```
PI: procedure
        return 3.141
```

To access the value you just have to make a call to the function "PI()" as in

```
        Area = PI() * (Radius ** 2)
```

INDENTATION

It has already been mentioned that text in comments should be indented. The code that appears in "if," "do," and "select" instructions should also be indented. A tab of three to eight characters is usually enough, with additional tabs for each level of nesting. This is to help you "visualize" the structure of the code and helps associate the various instructions to their nesting level.

"if," "else," and "select"

The "if" and "else" instructions allow you to associate multiple instructions with each, using the "do ... end" instruction. Always use the "do ... end" instructions as in the following layout

```
        if( something ) then do
                function1()
                end
        else do
                function2()
        end /* if */
```

```
        when ( Command == "LM_LOG" ) then do
                call OutputLogMsg(LogMsg)
                end
        otherwise do
                call OutputLogMsg( ,
                        "SysLog: Bad Cmd: " Command LogMsg)
                end
end
```

That finishes the main part of the logger host. Now let's take a look at that function we ran into earlier, "OutputLogMsg()". The purpose of "OutputLogMsg()" is to take the argument given to it and write the argument to the log file together with a time stamp, which consists of the date and the current time.

In ARexx the start of a function is identified by a label which is the the name of the function followed by a colon, ":". Good programming practice dictates that you should try to reduce the area from which a variable can be accessed; the technical term for this concept is limiting the "scope" of the variable. In an ARexx program all variables are global and can be accessed by all functions, unless the "procedure" instruction is used. To prevent "OutputLogMsg()" from having access to all the variables in the main part of the program, the program uses the following declaration for the function:

```
        OutputLogMsg: procedure expose LOGFILENAME
```

The "procedure" instruction prevents a function from having access to any variable in the main part of the ARexx program other than the variables in the "expose" list. So the only "global" variable which "OuputLogMsg()" can access is the variable "LOGFILENAME".

The parse instruction is used by "OutputLogMsg()" to obtain the argument which was passed to it.

```
        parse arg Msg
```

Anybody who has tried to create a file with ARexx knows that there is no "Create()" function. To create a file in ARexx, one has to be open the file in "Write" mode. Be carefu,l however. If the file already exists and you open it in "Write" mode, you will delete the previous contents. The following fragment just figures out in which mode the log file should be opened:

```
if Exists( LOGFILENAME ) then do
        OpenMode = "Append"
        end
else do
```

This format is less error prone and it makes it easier to just insert or delete an instruction to either branch.

"select"

Again with the "when" instructions use the "do ... end" instructions as in

```
        select
                when (Expression) then do
                        Function()
                        end
                when (Another Expression) then do
                        Function2()
                        end
                otherwise do
                        say "select failed"
                        exit 10
                        end
        end /* select */
```

When testing a variable against several alternatives use the "select" instruction over nested "if" instructions since the intention of the comparisons is easy to see and is less error prone than multi-nested "if" instructions.

One last word on the "select" command: Always include the "otherwise" branch and include it last. Even if you never expect the "otherwise" clause to be executed, include it and put in an instruction to output an error message. You'll be surprised how many times that error message will be displayed when developing your program!

MODES

A number of ARexx functions have arguments which are called "modes." A mode is a string in which only the first character is significant and it is used to select different options in a function call. Many programmers choose just to use the first character, as in

```
        say Show("L")
```

However, readability and understanding is better served if the entire string is listed, as in

```
        say Show("Libraries")
```

This leaves little doubt as to what we are interested in. A general programming rule is to be explicit; make your intentions as clear as possible.

FUNCTIONS

You should place a comment just before the start of each of your functions. This comment should include the function name, the type of arguments it accepts and what it will do. If it returns a value you should describe what it is that it will return.

In ARexx all variables are global, meaning that all the variables in the main part of your program can be accessed and modified by any of your functions. This is generally a bad practice; just think what would happen if your main program and one of your functions used the same name for a variable and you didn't notice. To prevent a function from having access to all the global variables, use the "procedure" instruction after the function label as in

```
        FuncName: procedure
```

This means that the variables in the main part of the program are distinct from the variables in your function. Try running the following example:

```
        AccessMode = "Write"
end
```

Finally, the log file is opened in the correct mode and the message together with a time stamp is written out;

```
if ( Open(LogFile, LOGFILENAME, AccessMode) ) then do
        call WriteLn( LogFile, Date('Ordered') Time() Msg
)
        Close(LogFile)
        end
```

That is the host completed, now lets turn to a simple example of how to log a message with the host from within an ARexx program. This ARexx program is called "log.rexx", listing 2. You can use it from within your Shell/Cli scripts to log messages with the logger host. One use might be to log when scripts are run and when they finish. To do this add the the following to your scripts;

```
        rx log ProgramName Starting
        /* rest of your script goes here */
        rx log ProgramName Finished
```

The first thing that "log.rexx" does is check the number of arguments that were supplied to it on the command line. If no arguments were given, an error message is printed and the program exits. Otherwise the parse command is used to fetch the argument and assign it to the variable Msg.

It is possible in ARexx to do error trapping. Thus, when an error is encountered, you can have ARexx jump to a label in your program. You could then evaluate the error and do the required clean-up operations before exiting. One problem we could encounter in our log script is that the host is not running. In ARexx this would generate a "syntax" error. We would like to output an informative error msg when this occurs so we will trap all syntax errors;

```
        signal on syntax
```

This line will cause the program to jump to the "syntax" label when a syntax error is encountered. The code to handle this error is quite straightforward;

```
syntax:
        say "Error : logger host not running"
        exit RC
```

```
/*
        stupid.rexx: a very silly program to count to print
out the numbers 1 to 10
*/

do Count = 0 to 9
        call MyPrintFunction(Count)
end
exit

MyPrintFunction:

parse arg Count
/*

Since the main program starts at 0 and we want to print
out the range 1-10 just add one to the argument sent by
the main program.

*/
Count = Count + 1
say Count
return
```

If you run the above program (rx silly), you will find that it does not print out the numbers from 1 through to 10. If, however, you change the line

```
        MyPrintFunction:
```

to

```
        MyPrintFunction: procedure
```

and try run the program again it does print out the numbers 1 through 10. In the first version the variable "Count" variable is shared by both the function "MyPrintFunction" and the main program. So as well as beeing incremented in the main program, it also gets incremented by one every time "MyPrintFunction" is called. By adding the procedure keyword, the "Count" variable in the "MyPrintFunction" becomes distinct from the "Count" variable in the main program. This is a simple example and the bug would be easy to spot, but if you start writing more complex programs, problems like this would be harder to find, so stamp the bugs out before they occur by following some rules.

You will find that sometimes a procedure will need access to a number of global variables which you don't wish to pass as arguments to the function; maybe the function needs to access the global variables that you are using as constants. If you feel that a function needs to have access to a variable in your main program, which you don't wish to pass as an argument, you should declare the variable in an "export" list as in:

```
        FunctionName: procedure expose GlobalVariable
GlobalVariableTwo
```

The above allows the function to access the only global variables "GlobalVariable" and "GlobalVariableTwo." So if the function made a reference to "GlobalVariable," ARexx knows that it is to use the "GlobalVariable" in the main part of the program and not to create a new "GlobalVariable" for the exclusive use of the function.

The above should help you create your own style rules. You may be wondering if I really follow mine; well you can review my code! Have a look at the ARexx code in the article "SysLog: An ARexx Host for Logging Messages." Did I follow all the above rules? Did you consider that by following the above guidelines I have actually made the code more readable, or not? Were there any other style rules that you think I followed in the article which are not mentioned above? It's possible, once you develop your own programming style and use it for a while, you almost forget it's there, but others can still see you have style! —JPG

RC is a special variable in which ARexx stores an error code after an error is detected.

When an ARexx script is run the default command port that it sends commands to is REXX. So the script has to change the command port to "SYSLOG", the logger's port. This is accomplished by using the "address" instruction; since the port name is stored in a variable, the "address value" form of the instruction is used.

```
address value LOGGERPORT
```

Now all commands are sent to the logger host. To send the logger host a "log" command together with the message, you need only say

```
'LU_LOG' Msg
```

To switch the command host back to REXX, we use another form of the "address" instruction;

```
address
```

This version of the "address" instruction switches back to the previous command host; if we were to use it again, we would switch back to the logger.

The above three commands could be written as one instruction, yet another version of the "address" instruction.

```
address SYSLOG 'LU_LOG' Msg
```

will send just the one command to the specified command host, without changing the current port. NOTE: In this form of the "address" instruction the name of the port must be stated explicitly; you cannot use a variable. For example, the following won't work:

```
address LOGGERPORT 'LU_LOG' Msg
```

This will instead try to send the command to a port named "LOGGERPORT". And the following will result in a syntax error:

```
address value LOGGERPORT 'LU_LOG' Msg
```

The program "stoplog.rexx" is quite straightforward and you should have no problems figuring it out for yourself. You should now know enough to add your own commands to the host. Try adding a command to copy the contents of the current log file, "s:syslog", to another file, "s:syslog.old".

If you have any questions on anything that I have presented above you can drop me a line c/o *Amazing Computing*, or if you have access to ACSnet or AARnet, my electonic mail address is paulg@tpird.tpl.oz.au and on Internet it is paulg@tps.com

BIBLIOGRAPHY:

ARexx User's Reference Manual by William Hawes, 1987
"Message Logger" by Brian Zupke, *Amazing Computing*, August 1991
Using ARexx on the Amiga by Zamara and Sullivan, Abacus, 1991

Listing 1

```
/
************************************************************
    Logger.rexx

    This is an Arexx Command Host which will add
messages
    sent to it to a the log file "s:syslog".
    Two commands are understood:
        LH_LOG Message   - the message is added to
the log file
        LH_STOP Message  - the message is added to
the log file
                              and then the logger
shutsdown.

    To run this host use:
          run <NIL: rx logger.rexx
************************************************************/
LOGFILENAME     = "s:syslog"  /* The name of the
log file */
OURPORTNAME     = "SYSLOG"     /* The name of our
port */

/*
   check to see if the rexxsupport library is
loaded.
   If not, try and load it.  If the load fails
exit, after
   logging some error msgs.
*/
if( ~ Show('Libraries', "rexxsupport.library") )
then do
   if( ~ AddLib("rexxsupport.library", 0, -30 ,0)
) then do
      call OutputLogMsg(
               "SysLog: ERROR : Cannot load
RexxSupport.library")
      exit 20
   end
end

/*
   Create a SYSLOG port.  If we cannot create a
port, log our
   failure and exit.
*/

if( ~ OpenPort(OURPORTNAME) ) then do
    call OutputLogMsg("SysLog: ERROR : Cannot
create port")
```



```
    exit 20
end

/* log a Startup msg */
call OutputLogMsg("SysLog: StartUp" )

do forever
   /* Wait for a Message Packet to arrive at our
port  */
   call WaitPkt( OURPORTNAME )
   Packet = GetPkt( OURPORTNAME )

   /*
      Retreive the Message out of the Packet,
      The first word is a "command" and the rest
      is a message to be logged
   */
   if( Packet ~= Null() ) then do
      parse value GetArg(Packet) with Command " "
LogMsg
      upper Command

      /*
         The client that sent this message does
         not care about any errors.  So we will
         return the packet now with a return
         code of 0
      */
      call Reply(Packet, 0)

     /* Figure out which command it is ... */
     select
        when ( command == "LH_STOP") then do
```

```
                call OutputLogMsg("SysLog:
ShutDown:" LogMsg)
                ClosePort( OURPORTNAME )
                exit 0
                end
            when ( Command == "LM_LOG" ) then do
                call OutputLogMsg(LogMsg)
                end
            otherwise do
                call OutputLogMsg(   ,
                        "SysLog: Bad Cmd: " Command
LogMsg)
                end
        end /* select */
    end /* if */
end  /* do forever */

/*
    the following line should never be executed.
However, it is
    good practise to include it just in case there
is a bug in
    the above code.
*/
exit 5
/
**************************************************************/
  End of main
/
**************************************************************/

/
**************************************************************
   OutputLogMsg( string ):  Will add the string to
the end of
                              the log file together
with a
                              timestamp.
**************************************************************/
OutputLogMsg: procedure expose LOGFILENAME
parse arg Msg

/*
      Make sure the log file exists if it doesn't
      then we create it.  The file is then opened,
      the msg is writen and the file is then
closed.
*/

if( Exists( LOGFILENAME ) ) then do
   AccessMode = "Append"  /* file exists, Append
to it */
   end
else do
   AccessMode = "Write"   /* file does not exist,
create it */
end

if( Open(LogFile, LOGFILENAME, AccessMode)) then
do
   call Writeln( LogFile, Date('Ordered') Time()
Msg )
   close(LogFile)
   end
else do
   exit 10
end
return
```

Listing 2

```
/*
  log  - sends a message to the logger host
         usage:  rx log msg
        where msg is the message to be logged by the
          logger host
*/

LOGGERPORT = "SYSLOG"  /* name of the logger host's
port */

/* was a message specified */
if( Arg() == 0 ) then do
   say "Error:  No message specified."
   say " Usage: rx log <string>"
   exit 5
end

parse arg Msg                   /* get the argument */

signal on syntax                /* trap errors */

address value LOGGERPORT        /* set command host to
SYSLOG */
'LH_LOG' Msg                    /* send the command */
address                        /* switch back to REXX */

exit 0
/
*************************************************************/
/
*************************************************************/
/*
  Trap on syntax error routine.  Simply, prints out
an
  error message.
*/
syntax:

say "Error: logger host not running"
exit 10
```


Listing 3

```
/*
  stoplog  - sends a shutdown command to the logger
*/

LOGGERPORT = "SYSLOG"

if( Show('Ports', LOGGERPORT) ) then do
    address value  LOGGERPORT /* make SYSLOG the
command host */
   'LH_STOP' "from StopLog"  /* send the command */
   address                   /* switch back to REXX
*/
end

exit 0
```

Please write to:
Paul Gittings
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Bitmapped Graphics

by Dan Weiss

Introduction

On the Amiga there is a well-defined standard for the exchange of bitmapped graphics, IFF ILBM. As robust as the standard is, it is still virtually unknown outside of the Amiga community. There may come a point where you will want your program to exchange bitmapped graphics with the "outside" world. If so, you will need to know what other formats are available, and what their strengths and weaknesses are. This article will look at several popular standards for various microcomputer platforms.

Picture Speak

As with most niches of technology, bitmapped graphics has its own vocabulary. Here then is a quick summation of the terms that are used in this article. A pixel, short for picture element, is the finest division of an image that can be specified. A bitmap, in the truest sense, is a computer structure where an on-bit defines a pixel of one color, and an off-bit defines a pixel of another color, usually black and white respectively. However, in current parlance, bitmap has become synonymous with pixmap. A pixmap is much the same as a bitmap except that multiple bits are used to determine the color of a pixel at a given point. This is done one of three ways, as grayscale, palette entry, or actual data. A grayscale picture, by definition, deals only with brightnesses of white. A grayscale picture will tell a program what range of values it will generate (0-1, 0-15, 0-256), then the program maps those values to ones it can handle. In a palette-entry picture, the data is not the actual colors, but rather indexes into a list of colors. In a 16-color picture on the Amiga only 4 bits of data (representing values 0-15) are stored for each pixel, but each of the sixteen colors is from a selection of 4096 colors (four bits each of Red, Green and Blue). This method greatly reduces the size of images. The final class of files stores that full actual data in the file. This method is usually used when there are more than 256 colors in a single picture. In the case of a 24-bit picture there would be no saving in having a color list like a palette picture because the index is the data. A reader is a program or piece of code that reads and understands a given file format. A writer writes a given format.

Finally, a short word about color theory. This article assumes that you understand that all colors shown on computers are made up of differing amounts of red, green and blue light. Since this is so, all current bitmapped graphics standards revolve around this. But, on the horizon are bitmapped formats that revolve around the cyan-yellow-magenta-and-black color model used by printers everywhere.

A look at bitmapped graphics formats

IFF ILBM - Interchange File Format InterLeaved Bit Map

Before other formats are examined, a frame of reference is needed. Since most readers of this article are familiar with IFF ILBM, it will serve as the standard. IFF ILBM was originally designed by Electronic Arts for their *Deluxe Paint* program. It has since been adopted, along with the rest of IFF, as the standard for the Amiga. As originally defined it can handle up to 256 bitplanes of data. In the case of the Amiga, there was a practical limit of 5 bitplanes, since that is all the Amiga directly supported. The format uses a color-lookup table (palette) called a CMAP in which each entry can hold a byte of color information each for red, green and blue. The format is somewhat limited by having only a simple RLE compression scheme, and no way to specify pixel size, although it can specify pixel aspect ratio. (See sidebar on compression standards at the end of this article.)

For the Amiga, the specification was extended to handle 64 colors with only 32 palette entries by using special "halfbrite" colors. It was also extended to handle the Amiga's special "Hold And Modify" (HAM) mode that allows for pseudo 12-bit (4096) color. These modifications allowed the format to apparently handle more complex images than it actually did, and at noticeable savings of disk space and memory. But these solutions are very machine specific and somewhat scorned by the rest of the industry. Recently, support for "deep" bitmaps, like those with 12 and 24 bitplanes, has been introduced. To date, however, no modifications have been made to address the limitations in compression and pixel size definition.

On many levels the IFF ILBM format is a strong one. The IFF structure is easily expanded and modified. The only reason the weaknesses of the format have not been addressed is due to a reluctance to "break" non-conforming readers. On the Amiga, this is the format of choice.

GIF - Graphics Interchange Format

GIF (pronounced JIF according to the specification, although I have never heard it pronounced that way) was developed by CompuServe for "defining a mechanism for the storage and transmis-

sion of raster-based graphics information." This format is perhaps the most widely supported format of all, in terms of computer platforms. It readily supports 256-color palette pictures with 8 bits of R, G, & B information for each color. In this respect it is similar to the IFF ILBM specification. It differs, however, in a number of other ways. GIF uses LZW (Lempel-Ziv-Welch) compression. This is a powerful compression method that modifies itself according to the picture, and is very efficient. The format also allows for multiple pictures per file, but this feature is rarely used.

When GIF was created, one of the design ideas was to use it as a method of storing files so that they could be examined as they were being downloaded—remember that format was designed by CompuServe. To optimize this, some files are stored interlaced. An interlaced file stores the scan lines of the file in a mixed order. In effect every nth line is stored sequentially. This was done so that the user could tell what the picture was when only a portion of the file had been transmitted.

GIF is a good format for exchanging bitmapped pictures that are meant to viewed more than printed. There are PD GIF viewers for just about every computer platform, but not many pieces of commercial software read GIF files.

IMG - GEM Image Format

The IMG format has its origins in the GEM operating system from Digital Research Inc. (DRI). This is the operating system used by the Atari and by some programs on the IBM PC, most notably Ventura Publisher. While it is thought of as an exclusively black-and-white format, in its original specification it is defined as a RGBW (Red, Green, Blue and White) format. Currently there is a resurgence of interest in a Color IMG format on the Atari, but no one standard has risen to the top. For all intents and purposes, IMG is a B &W format.

As a B&W format, it is very powerful, and reasonably efficient. While the format only supports a simple RLE encoding, it has virtually no header—only 16 bytes—compared to a TIFF file. Also the IMG format defines the size of the image's pixels in microns. This is extremely important. One of the common uses of the IMG format is for scanned images. Today it is not uncommon to have a collection of images scanned at different resolutions for different needs. While you may know that one image was scanned at 200 dpi and another at 300 dpi, the computer has no way to know unless the file tells it. This is a major problem with saving scans in IFF ILBM format. Of course, pixel aspect ratio is implied by the pixel size, so in this way IMG equals, and surpasses IFF.

While limited to B & W images, this format is nonetheless alive and well on several computers. If you are dealing with monochrome scans and want a good format, but not the work associated with TIFF, IMG is a good bet. This format is supported as at least an option by many image file converters on microcomputer platforms.

PCX - PC Paintbrush Format

The PC Paintbrush format is a good example of the Peter Principle applied to a graphics file format. It is hard to say anything good about the format other than it is widely supported on the MS-DOS platform. Like several of the formats in this article, including *MacPaint* and IMG, PCX is in wide use because it was one of the first formats available on its platform.

When the format was created, the designers seemed to think only in terms of existing hardware. Scanners, B & W, let alone color, were scarce, and the existing display hardware was limited. PCX's compression, RLE, does best when dealing with long runs of the same color, common in hand-drawn pictures, but rare in scans. At the time 16 colors was also thought to be more than enough, so a 16-color

Sidebar: Compression Methods

by Dan Weiss

Compression Methods

Virtually every bitmap image file format uses some form of compression. There are two good reasons for this. First bitmap images can be huge, and furthermore they tend to compress rather well. File #5 in the compression comparison chart shows that both the sizes of files and the compression achieved can be amazing. Each of the formats discussed in the article uses one or more of the following techniques to reduce the size of their files.

RLE - Run Length Encoding

Many times a bitmap image is made up of large spaces that are the same. It could be the white background for a drawing, or the blue sky in a photograph. If the computer sees such a run of pixels that are the same it will try to compress it. With RLE, the computer will first store an indicator code and a repeat number, then the byte that is to be repeated. This method usually catches runs of the same color. But, if the data in the byte is the same as the next, as in an ordered dither or pattern, this format will also compress mixed color data. Depending on the maximum run the implementation can handle and the nature of the data, this can be a very good or very poor compression technique. This method is sometimes also known as PackBits. IFF ILBM, GIF, IMG, PCX, MacPaint, PICT, and TIFF all use this method of compression.

LZW - Lempel-Ziv-Welch

LZW goes a step beyond RLE in that it can encode several bytes at once. Each time a byte is read in it is added to a string of the bytes that came before it and given a code. If at anytime the computer can look ahead and see a series of bytes that matches a string it has already encountered, it will use the code instead. This method catches the longer term patterns that may emerge in a scanned image. Some files, such as File #6 on the compression comparison chart, show dramatic results with this method, while others, like file number 5, show little. This method can be further improved by using a predictor. This is a mathematical function that is first applied to the data before it is compressed. A common predictor is a horizontal differencing predictor. The predictor replaces each byte with the amount it is different from the last in the original image. In this way it is the change itself that is being compressed. Due to the computation involved in handling LZW files they tend to compress and decompress much slower than RLE files. GIF and TIFF both use this method of compression, though only TIFF uses predictors.

palette was placed in the format. With this 2-, 4-, 8- and 16-color files could be easily stored. Since then, in response to new hardware, the format has been extended to support 256-color pictures.

Unfortunately, since no provision was made to expand the format, the color palette was placed after the end of the file. As in the past, the designers of the format made some assumptions about the 256-color extension. The logic was that if you can have 256 colors, who would want less. Therefore any file over 16 colors is always stored as a 256 color file. This means that a 32-color file is very inefficient because it has to carry around three unused bitplanes. Even though PC hardware that existed when the format was created only supported a handful of colors, the designers did have the forethought to specify each color in the palette with a byte each for R, G &B for a total of 24 bits of information.

As I have said, the main strength of the format is that it is widely supported on the PC. If your need is to exchange small files with programs that don't support better formats, then PCX is okay. In general, because of its lack of flexibility and poor compression, I would avoid this format.

MacPaint - MacPaint File Format

The MacPaint file format is tied very tightly to the original Macintosh and its hardware design. It is monochrome only, 72 dpi pixels only, and always 576 x 720 pixels. The reasons for this are simple; the original Macintosh was monochrome only, and the display and printer were both 72 dpi. The static file size made the file very easy to compress and decompress using statically defined memory structures that are relatively small (>52K). Also to aid anyone using this format, Apple built the compression software into the ROM. It is a simple matter to write a short program that reads and writes compressed MacPaint files—on the Macintosh that is. For the rest of us, the format of the file and the compression method, RLE, are reasonably well documented in non-Macintosh publications.

There is little reason to write files in this format. It is very limited since the document size is fixed, and the resolution is assumed to be 72 dpi. However, it may be a good format to read. On BBSs and PD disks across the world there exists thousands of megabytes of MacPaint files. One popular CD-ROM alone has over 30 MBs of MacPaint files. These files contain just about every type of clip art you can imagine and then some. MacPaint files can often serve as a foundation for a larger graphic, a brush, or some quick clip art. When used as 300 dpi graphics, they can be quite sharp, and of course, quite small (1/4 original size).

PICT - Macintosh Picture Format

Like MacPaint, the PICT format is tied tightly to the Macintosh and is used virtually nowhere else. As with MacPaint, I would suggest this as a format to read, but not write. The similarities with MacPaint end here. The PICT format is first and foremost a vector drawing format. It was designed to encompass every drawing command available to the Macintosh. In this way anything that could be drawn to the Macintosh screen could be stored in a file. This of course includes bitmaps. In the original PICT format, only B & W bitmapped images were supported. Because of the original limitations of the Macintosh, the PICT format often stores a picture as a series of smaller pictures. Also, to be sure that the picture is imaged correctly it may first draw the image once as a mask, in white to clear an area for the picture, then a second time to lay down the image. This means over half the information in a PICT file may be useless to you.

CCITT Group 3 - Fax Compression

The Group 3 format was specifically designed for compressing fax images. It takes a radically different approach from the two methods above. Studies were done on typical faxes, and the most common white and black pixel (bit) runs were determined. This information was placed in a table and codes were assigned for each run. Since the table is the same no matter what, it is built into every fax machine. The success of the method comes from two areas, first because it is designed for the noisy data that is common with faxes and additionally because it compresses at the bit, not byte, level. When decompression is handled directly by special chips, the speed can be incredible, but when handled it software it can be slow since it works at the bit, and not the byte, level. Only TIFF uses this method, and it is suitable only for B&W images. —DW

Compression Comparison Table

File #	1	2	3	4	5
Number of Colors	2	16	256	16.8	16.8
Actual Size	52	64	92	439	900
IFF	32	12	96	431	163
GIF	26	5	42	-	-
IMG	38	-	-	-	-
PCX	45	12	94	-	-
MacPaint	33	-	-	-	-
PICT	34	34	76	443	122
TIFF	26	6	44	409	67

All file sizes are rounded and shown in KiloBytes.

Bitmapped Graphics Format Comparison Table

File Type	1	2	3	4	5	6	7	8	9	12	16	24	32	Compression Methods	Pixel Size	Comments
	Bitplanes Supported															
IFF	●	●	●	●	●	●	●	●	●	●	●	●	●	RLE	No	Use on the Amiga
GIF	●	●	●	●	●	●	●	●						LZW	No	Widely Accepted
IMG	●													RLE	Yes	B&W Scans
PCX	●	●	●	●				●						RLE	No	For PC Compatibility
MacPaint	●													RLE	No	Source of Images
PICT	●	●		●				●						RLE	Yes	For MAC Compatibility
TIFF	●	●	●	●	●	●	●	●	●	●	●	●	●	LZW,RLE CCIT Group 3	Yes	Format of Choice

With the introduction of color Macintoshes, a better solution was needed, and PICT 2 was born. PICT 2 is an extended version of PICT, but is designed to serve the same purpose. With the new version came support for color bitmaps. Images from 1 to 32 bits can now be stored in the PICT 2 format. Much like the PCX format, PICT limits your choices of bitmap depth to 1, 2, 4, 8, 16 and 32 planes. Notice, that each of these is a number that fits quite well into a byte or multiple of bytes. In the case of the 16- and 32-bit files, the "extra bits" (1 and 8 respectively) are used as an Alpha channel. An Alpha channel is used to store extra information pertinent to each pixel, like transparency, that is not directly related to its color.

The version of RLE compression used in this format is reasonably good and is always applied, making for manageable files. As a 24-bit color standard, PICT 2 is really only of use on the Macintosh. Its most notable features are that it stores Alpha channel data and contains pixel size information. There are some very interesting 32-bit color files in this format, but certainly not as many as there are MacPaint files. In general the only reason to heavily support this format is to easily exchange images with the Macintosh.

TIFF - Tagged Image File Format

While the formats have been reviewed in alphabetical order, the best has definitely been saved for the last. TIFF is the most comprehensive, and most complex, bitmapped file format that this article covers. TIFF, like the IFF standard, is designed to be expanded. Instead of the FORM/CHUNK metaphor used by IFF, TIFF relies on tags that make up the header of the file. A TIFF tag is made up of a tag number, data type number, data count, and data offset pointer. The tag number is a distinct, registered number that identifies the tag. The field type defines the kind of data referenced by the tag. The count is the number of times that the indicated data type will be repeated. The offset is a pointer into the body of the file, where the data referenced by the tag is located. In cases where the data is small, it will be stored in the offset pointer. This makes the many two-byte tags efficient.

Because of the structure of the tag, again like IFF, it is easy to skip over those tags that you do not understand. Several of the tags such as Artist, DateTime, Host Computer, Software, Make, and Model have no bearing on the image at all. The tags just mentioned are very useful, however, when you are having a problem. As a developer you can look at a file and determine who and what created it. Often this can be instrumental in solving any problems you may have with the file. The tag-based system also allows a program to support TIFF to the degree that makes sense for it. If the program stores monochrome scans, then the many tags concerning color are unnecessary. Yet, though the program may support a only fraction of the tags in the format, it is still a valid file. In most cases you have to support all aspects of a format to support any of it.

The difficulty of TIFF is being a comprehensive reader. Since there are so many tags, and so many ways to store and interpret the data in a file, a reader can be quite complex. Adding to the complexity is the fact the TIFF currently supports at least three different compression schemes in addition to no compression at all. The LZW compression can also be subject to a predictor. The standard is equally complete in the kinds of pictures it handles: B&W, grayscale, palette and true color. It even has a tag for the all-important pixel size.

If you are serious about transferring files between many different platforms, this is as close as you can come to a universal format. Because of the complexity of the format, I would recommend testing it far more extensively than you would test any other such capability.

From Here

There are of course many other bitmap graphic formats. The seven formats discussed in this article were chosen because they are widely used as a method of exchange in their respective niches or platforms. Other formats range from the practical, Targa, to the out-of-date, Degas. One of the formats discussed above will more than likely meet your needs. The information in this article is insufficient to write any sort of file handler; that was never the intent. If there is interest in code-level discussions of these formats, this publication is a good venue for such articles. In the meantime, listed below are sources of information for the formats discussed above. Good luck in dealing with the outside world.

Please write to:
Dan Weiss
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Three of life's essentials:

Amazing Computing For The Commodore AMIGA

Amazing Computing For The Commodore Amiga is dedicated to Amiga users who want to do more with their Amigas. From Amiga beginners to advanced Amiga hardware hackers, AC consistently offers articles, reviews, hints, and insights into the expanding capabilities of the Amiga. *Amazing Computing* is always in touch with the latest new products and new achievements for the Commodore Amiga. Whether it is an interest in Video production, programming, business, productivity, or just great games, AC presents the finest the Amiga has to offer. For exciting Amiga information in a clear and informative style, there is no better value than *Amazing Computing* .

AC's TECH For The Commodore AMIGA

AC's TECH For The Commodore Amiga is the first disk-based technical magazine for the Amiga, and it remains the best. Each issue explores the Amiga in an in-depth manner unavailable anywhere else. From hardware articles to programming techniques, *AC's TECH* is a fundamental resource for every Amiga user who wants to understand the Amiga and improve its performance. *AC's TECH* offers its readers an expanding reference of Amiga technical knowledge. If you are constantly challenged by the possibilities of the world's most adaptable computer, read the publication that delivers the best in technical insight, *AC's TECH For The Commodore Amiga.*

AC's GUIDE To The Commodore AMIGA

AC's GUIDE is a complete collection of products and services available for your Amiga. No Amiga owner should be without *AC's GUIDE*. More valuable than the telephone book, *AC's GUIDE* has complete listings of products, services, vendor information, user's groups and public domain programs. Don't go another day without *AC's GUIDE*!

100% of the recommended daily allowance of Amiga information.

1-800-345-3360

Transformer

by Laura M. Morrison

Transformer makes it easy to use changes of size and angle in your Amiga Graphics. Unlike paint programs which use brushes and rubber-band boxes for re-sizing and rotating, Transformer works on images. It will accept, and work on, an image up to 1024 x 1024, and never balk as Paint programs may at the size of the brush, or require you to execute the action "blind" with only the outline of the rectangular brush showing. Because Transformer takes its sizing information from an ASCII file, and not from a transient rubber-band box, it can scale a sequence of images by exactly the same amount and rotate images an exact angle. It rotates without distorting the image. Except for HAM images it reproduces an image's palette exactly.

Transformer reads an image, transforms it according to your instructions, and saves the new image. The program gets its instructions by reading an ASCII parameter file, called 'transparams' which you can prepare using 'ed.' In this parameter file you describe exactly how the image is to be transformed by providing:

1. The depth, width, and height of the input image to be transformed. If otherwise unknown, this information can be obtained by running 'view' on the image. The depth, width, and height should be input as integers.

2. Translation amounts along the x-axis and y-axis. Use 0.0 and 0.0 if no translation is desired. You can use translation to ensure that the transformed image plots on the screen. Although the translation amounts refer to pixels, they must be input as floating point numbers.

3. Scale factors for the x-axis and y-axis. Use 1.0 and 1.0 if there is to be no change of size. The scale factors can be floating point, even for increasing size. Transformer increases without making an image coarse, leaving any filling in, if necessary, to the artist.

4. Angle of rotation in radians. This may be any angle from 0 to 360 degrees. (360 degrees = 2*3.1416.. radians.) Rotation is about a line perpendicular to the screen at the fixed point and goes counter-clockwise. The angle is a floating point number.

5. A fixed point of the transformations. Pick a point that ensures the transformed image will plot on the screen. It can be an estimate, or a paint program will give you exact coordinates. The fixed point is relative to the input image and not the screen. Note that although the fixed point refers to pixels, it should be input as a floating point number.

6. Number of rotations to do (integer) and the angle increment in radians (floating point). Transformer will do a sequence of rotations of angles increasing by the increment. It saves each image as it goes.

Illustration (a) is the Migraph scan of the top of a pencil drawing. Illustration (b) is a composite of (a) and two other similarly scanned parts of the same drawing. The parts were each scaled 0.57 and the resulting images combined to produce (b). Illustration (c) is the scan (Illustration (a)) reduced 0.325. Illustration (d) is the same scan reduced by 0.325 and rotated 1.5704 radians.

The parameters for the transformations used to produce the illustrations, as they appear on the 'transparams' file are:

Illustration (a):

```
1 640 600 0.0 0.0
1.0 1.0 0.0 0.0 0.0 1 0.0
```

An image of depth 1, width 640, and height 600 is to be reproduced without change of size and with no rotation. The fixed point is the upper left corner of the screen.

Bottom parts for Illustration (b) (not shown):

```
1 640 600 0.0 -200.0
1.0 1.0 0.0 0.0 0.0 1 0.0
```

The scanned image is to be shifted up on the screen 200 rows to give a copy of the bottom of the 600 row scan-image.

Each of the three parts of Illustration (b) (shown reduced and already composed):

```
1 640 600 0.0 0.0
0.57 0.57 0.0 0.0 0.0 1 0.0
```

The image is reduced by a scale factor of 0.57 (chosen because it makes an image that fits exactly in the screen frame).

Illustration (c):

```
1 640 600 0.0 0.0
0.325 0.325 0.0 0.0 0.0 1 0.0
```

The scanned image is reduced by a scale factor of 0.325.

Illustration (d):

```
1 640 600 100.0 0.0
0.325 0.325 1.5704 190 189 1 0.0
```

The scanned image, reduced by a scale factor of 0.325, is to be shifted 100 pixels to the right and rotated 90 degrees, that is, 1.5704 radians. Since there is a rotation the Transformer sidesteps the distortion introduced by the non-square pixel by first transforming the image to an inches coordinate system, then rotating in terms of inches, and finally, transforming back into screen pixel coordinates for plotting on screen. X was translated 100 pixels to the right, and the fixed point was taken in the middle of the screen, so the resulting image would be well onto the screen. Without the shift in x, the head of the image might be off the left of the screen. You can guess an approximate fixed point, or a paint program can give you exact coordinates. The fixed point ensures the transformed image will plot on screen, but does not otherwise influence the image.

Not illustrated:

```
4 640 400 0.0 0.0
1.0 1.0 0.0 320.0 200.0 20 0.2
```

A high resolution, interlace image is to be rotated 20 times in increments of 0.2 radian. The rotation is to be about the center of the screen.

Transformer knows what variable it is reading by the variable's relative position in the input, so it is important to fill each variable position with a dummy value even though that variable will not be used. Also, the program requires floating point where it expects floating point, and integer where it expects integer.

Transformer runs from the CLI and not the WorkBench. To run the program enter the CLI and, at the CLI prompt type:

```
Transformer     <filename_for_output_image>
<input_image_filename>
```

In the same directory, besides the transparams file, you will need Raw2ILBM and ILBM2raw, Commodore's public domain conversion programs. If you do not have a copy of 'Execute' in your 'c' directory you will need that also. You will also need sufficient disk space to write the intermediary raw file and the final image file. Transformer reads the image to be transformed onto a CustomBitMap. Please be patient as this takes a while because Transformer first converts the input image to raw bitmap format using Execute and ILBM2raw. Only the upper left part of this CustomBitMap can be seen on screen. Facility for scrolling the CustomBitMap has not been implemented, but you can gain access to off-screen parts of the input image by translating the off-screen portions onto the screen and making an intermediary copy if necessary.

After producing the transformed image Transformer writes a raw image file, then, using Execute and 'Raw2ILBM', converts the file to ILBM format.

Transformer keeps a record of its activity on a file called 'Run_Notes.' You may want to record date and time information here also.

Programming code explanations are included as comments in the LISTING.

Transformer can be used for a variety of size and rotation transformations. Because of its repeatability and precision it works well with images scanned in parts and then combined. It makes shrinking spin-outs fast and easy. You can draw all your animation figures the same size, then, to make them recede in the distance, shrink them methodically smaller and smaller.

Although it is configured for transforming high resolution interlace images, you can change the resolution by changing WIDTH and HEIGHT (and the input and output file conversion information) and compile versions for each resolution you use. For example, to convert to low resolution 'hi1' and 'hi 1' must be changed to 'lo5' and 'lo 5'. (Note the space in the second string.) You will want to increase the precision if you wish to use the programs for technical or engineering applications.

Biographical Sketch

Laura M. Morrison has a Masters Degree in Mathematics from New York University, N.Y. She has worked as an Operations Research Analyst, designing applications software for Esso R&E, Union Carbide, and Eastern Airlines. She is at present self-employed.

LISTING - Transformer.c

```c
/* LISTING - Transformer.c
Copyright 1992 by Laura M. Morrison */
#include "stdio.h"
#include "libraries/dosextens.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "exec/exec.h"
#include "math.h"
#define WIDTH   640L
#define HEIGHT  400L
USHORT cmap[16];
SHORT *sa[9];
extern struct RastPort *rp1;
extern struct RastPort *rp2;
extern struct Screen *OpenScreen();
struct FileHandle *Open();
/* Set up information to be used later when opening the screens.
Leave DEPTH = NULL to be filled in later when 'indepth' is read
from the 'transparams' file. */
struct NewScreen newscr1 = {
        0,0,WIDTH,HEIGHT,NULL,0,1,
        HIRES|LACE,
        CUSTOMSCREEN|CUSTOMBITMAP,
        NULL,
        " Converting input image to the format ...",
        NULL,NULL };
struct Screen *scr1;
struct NewScreen newscr2 = {
        0,0,WIDTH,HEIGHT,NULL,0,1,
        HIRES|LACE,
        CUSTOMSCREEN|CUSTOMBITMAP,
        NULL,NULL,NULL,NULL };
struct Screen *scr2;
struct BitMap *bitm1;
struct BitMap *bitm2;
struct RastPort *rp1;
struct RastPort *rp2;
struct ViewPort *vp1;
struct ViewPort *vp2;
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
static float xs, ys, xt, yt;
static float xf, yf, xsl, ysl;
static float xrw, yrw, xu, yu, xr, yr;
static float xrv, yrv, xrs, yrs;
static float x, y;
static float xxs, yys, xxt, yyt;
static float za, cosza, sinza;
static float tanx, tany, delta;
static float u, v, uu, vv, xincr, yincr;
static char outfile[120];
static char gopic[120];
static char goraw[120];
static char str[80];
static char scr[80];
static char infile[120];
static char delfile[120];
main(argc, argv)
int argc;
char *argv[];
{
        LONG err, brr;
        LONG nextx, nexty;
        int row, col, cc, i, number, maxrot;
        FILE *fopen(), *pp, *ip;
        LONG nbr, success, colorsize, numcols, planesize;
        LONG indepth, inheight, inwidth;
if (argc < 3 )
{
        printf(" CLI Usage: Need name for transformed ");
        printf("output image and name of input image\n");
        printf(" on the command line.\n");
        exit(10);
}
IntuitionBase = (struct IntuitionBase *)
OpenLibrary("intuition.library",0);
if(IntuitionBase == NULL)
{
        printf("Couldn't open Intuition library\n");
        exit(10);
}
GfxBase = (struct GfxBase *)
OpenLibrary("graphics.library",0);
if(GfxBase == NULL)
{
        printf("Couldn't open Graphics library\n");
        CloseLibrary(IntuitionBase);
        exit(10);
}
printf("\n\n\n\n\n\n\n\n\n");
printf("                    ");
printf(" TRANSFORMER PROGRAM\n\n");
printf("                    ");
printf("Copyright 1992 by Laura M. ");
printf(" Morrison\n\n\n");
printf("\n\n\n\n\n\n\n\n\n\n");
Delay(100);
if (( pp = fopen("transparams", "r")) == NULL)
{
printf(" Need a parameter file, 'transparams', with\n");
printf(" depth, width, height of the input image;\n");
printf(" x and y translation amounts;\n");
printf(" x and y scale factors;\n");
printf(" the angle of rotation in radians;\n");
printf(" coordinates of the fixed point;\n");
printf(" the number of times to rotate");
printf(" and the angle increment between rotations.\n");
CloseLibrary(GfxBase);
CloseLibrary(IntuitionBase);
exit(10);
}
/* Keep a copy of the following scanfs on the 'transparams' file
for reference. */
fscanf(pp," %ld %ld %ld ",&indepth,&inwidth,&inheight);
fscanf(pp," %f  %f  %f  %f ",&xxt,&yyt,&xxs,&yys);
fscanf(pp," %f  %f  %f ", &za, &xf, &yf);
fscanf(pp," %d  %f  %f ",&maxrot, &delza );
fclose(pp);
/* Since the input is on a permanent file and less likely to be
in error than if keyed in, only this data check is included. */
if((za < 0.0)||(za > 6.2836))
{
        printf(" Need angle of rotation >= 0.0 and <= 6.2836
radians.\n");
        CloseLibrary(GfxBase);
        CloseLibrary(IntuitionBase);
        exit(10);
}
/* An exponential function may not be provided for C */
numcols = 1;
        for ( i = 0; i < indepth ; i++)
        {
                numcols = 2*numcols;
        }
colorsize = 2*numcols;
/* Set up a CustomBitMap and its screen for the input image.
Allocate memory for the structure and bitmap planes. Initialize
the structure, attach the bitmap to the screen and open the
screen.*/
planesize = (inwidth/8)*inheight;
bitm1 = (struct BitMap *)
AllocMem(sizeof(struct BitMap),MEMF_CHIP);
```

```
		if (bitm1==NULL)
		{
		printf("No memory for bitmap struct. \n");
		goto quit;
		}
InitBitMap(bitm1,indepth,inwidth,inheight);
for (i = 0; i < indepth; i++)
{
		bitm1->Planes[i] = (PLANEPTR *)
AllocRaster(inwidth,inheight);
		if (bitm1->Planes[i] == NULL)
		{
		printf("No memory for bitmap.\n");
		goto quit;
		}
		BltClear(bitm1->Planes[i],planesize,1);
}
newscr1.Depth = indepth;
newscr1.CustomBitMap = bitm1;
scr1 = (struct Screen *)OpenScreen(&newscr1);
		if (scr1==NULL)
		{
		printf("scr1 was null\n");
		goto quit;
		}
rp1 = &scr1->RastPort;
vp1 = &scr1->ViewPort;
/* Convert the input file to raw format. */
strcpy(gorow,"ilbm2raw ");
strcat(gorow,argv[2]);
success = Execute(gorow,0,0);
		if(success==NULL)
		{
		printf("Couldn't convert input image file.\n");
		goto quit;
		}
/* Open the raw image file and read it onto the bitmap. */
itoa(indepth,str1);
strcpy(infile,argv[2]);
strcat(infile,".hi");
strcat(infile,str1);
bfh = Open(infile,MODE_OLDFILE);
		if (bfh == NULL)
		{
		printf("Can't open image input file.\n");
		goto quit;
		}
		for ( i = 0 ; i < indepth; i++)
		{
		nb = Read(bfh,bitm1->Planes[i],planesize);
		}
cols = &cmap[0];
nb = Read(bfh,cols,colors*2);
Close(bfh);
LoadRGB4(vp1,cmap,16);
/* Delete the raw version of the input file. */
strcpy(delfile,"delete ");
strcat(delfile,argv[2]);
strcat(delfile,".hi");
strcat(delfile,str1);
success = Execute(delfile,0,0);
/* Set up a CustomBitMap and screen for the output image. */
		bitm2 = (struct BitMap *)AllocMem(sizeof(struct
BitMap),MEMF_CHIP);
		if (bitm2==NULL)
		{
		printf("No memory for bitmap struct. \n");
		goto quit;
		}
InitBitMap(bitm2,indepth,WIDTH,HEIGHT);
planesize = (WIDTH/8)*HEIGHT;
for (i = 0; i < indepth; i++)
{
		bitm2->Planes[i] = (PLANEPTR *)
AllocRaster(WIDTH,HEIGHT);
		if (bitm2->Planes[i] == NULL)
		{
		printf("bitmap2 planes were null\n");
		goto quit;
		}
BltClear(bitm2->Planes[i],planesize,1);
}
newscr2.Depth = indepth;
newscr2.CustomBitMap = bitm2;
scr2 = (struct Screen *)OpenScreen(&newscr2);
		if (scr2==NULL)
		{
		printf("scr2 was null\n");
		goto quit;
		}
rp2 = &scr2->RastPort;
vp2 = &scr2->ViewPort;
LoadRGB4(vp2,cmap,16);
number=0;
/* Open an ASCII file to collect data on program activity. */
fp = fopen("Run_Notes","a");
/* 'xinch' is inches per pixel horizontally; 'yinch' is inches per
pixel vertically. Values are based on 4:3 width/height screen
aspect ratio. */
xinch = 0.0125;
yinch = 0.015;
/* Multiply by 100 to enhance calculation precision. */
facx = 100.0*xinch;
facy = 100.0*yinch;
/* The fixed point after translation, in pixels. */
xo = xf + xxt;
yo = yf + yyt;
/* The fixed point in inches. */
uo = facx*xo;
vo = facy*yo;
/* Save recalculating this constant. */
cxf = xf*(1-xxs);
cyf = yf*(1-yys);
rotemore:
number++;
SetRast(rp2,0);
/* Evaluate these only once for each angle rotated. */
cosza = cos(za);
sinza = sin(za);
for (row = 0; row < inheight; row++)
{
		for(col = 0; col < inwidth; col++)
		{
		cc = ReadPixel(rp1,col,row);
/* Ignore pixels of background color. */
		if(cc != 0)
		{
				x = (float)col;
				y = (float)row;
/* Scale */
				if((xxs != 1.0)||(yys != 1.0))
				{
				xs = x*xxs + cxf;
				ys = y*yys + cyf;
				x = xs;
				y = ys;
				}
/* Translate */
				if((xxt)||(yyt))
				{
				xt = x + xxt;
				yt = y + yyt;
				x = xt;
				y = yt;
				}
/* Rotate */
				if(za)
				{
/* u and v are the point in inches coordinate system. */
				u = facx*x;
				v = facy*y;
```

```
/* Rotate in inches coordinates to preserve measurements and avoid
the distortion introduced by non-square pixels. */
                    xr = uo + (u - uo)*cosza  + (v-vo)*sinza;
                    yr = vo + (u - vo)*cosza - (u-uo)*sinza;
/* Translate back to pixels coordinates for plotting on the screen.
*/
                    x = xr/facx;
                    y = yr/facy;
          }
          nextx = (LONG)x;
          nexty = (LONG)y;
          SetAPen(rp2,cc);

if((nextx>0)&&(nextx<WIDTH)&&(nexty>0)&&(nexty<HEIGHT))
          {
          WritePixel(rp2,nextx,nexty);
          }
          } /* end of cc not zero loop */
          } /* end of col loop */
          } /* end of row loop */
/* Finished with this rotation now write an output image. */
finish:;
/* Number the output images so they don't overwrite one another.
*/
itoa(number,str);
SetAPen(rp2,1);
Move(rp2,30,30);
Text(rp2,str,strlen(str));
strcpy(outfile,argv[1]);
strcat(outfile,str);
ofh = Open(outfile,MODE_NEWFILE);
if (ofh != NULL)
{
          for ( i = 0; i < indepth; i++)
          {
          nb = Write(ofh,bitm2->Planes[i],planesize);
          }
          cols = &cmap[0];
          nb = Write(ofh,cols,colorsize);
          Close(ofh);
}
Delay(20);
strcpy(gopic,"raw2ilbm ");
strcat(gopic,outfile);
strcat(gopic," ");
strcat(gopic,outfile);
strcat(gopic,".pic hi ");
strcat(gopic,str);
success = Execute(gopic,0,0);
strcpy(delfile,"delete ");
strcat(delfile,outfile);
success = Execute(delfile,0,0);
fprintf(lp,"\n %s %s %s ", argv[0], argv[1], outfile);
fprintf(lp," %s \n ",argv[2] );
fprintf(lp," %ld %ld %ld \n",indepth,inwidth,inheight);
fprintf(lp," %4.2f %4.2f ",xxt, yyt);
fprintf(lp," %4.3f %4.3f ",xxs, yys);
fprintf(lp," %2.4f ",za);
fprintf(lp," %4.0f %4.0f \n ", xf, yf);
fprintf(lp," facx = %4.4f facy = %4.4f ",facx, facy);
za = za + delza;
/* If more rotations are required go back and start over. */
          if((za < 6.3)&&(number < maxrot))
          {
          goto rotemore;
          }
quit:;
/* Now close files, screens and release bitmap memory. */
if(lp) fclose(lp);
if (scr1) CloseScreen(scr1);
for ( i = 0; i < indepth; i++)
{
          if(bitm1->Planes[i])
          {
          FreeRaster(bitm1->Planes[i],WIDTH,HEIGHT);
          }
}
```

```
if (bitm1)
{
FreeMem(bitm1,sizeof(struct BitMap));
}
if (scr2) CloseScreen(scr2);
for ( i = 0; i < indepth; i++)
{
          if(bitm2->Planes[i])
          {
          FreeRaster(bitm2->Planes[i],WIDTH,HEIGHT);
          }
}
if (bitm2)
{
FreeMem(bitm2,sizeof(struct BitMap));
}
if (IntuitionBase)
          CloseLibrary(IntuitionBase);
if (GfxBase)
          CloseLibrary(GfxBase);
} /* end of main */
/* itoa is from Kernighan and Ritchie C Programming */
itoa(nr,str)
char str[];
int nr;
{
   int c, i, j, sign;
   if ((sign = nr) < 0)
      nr = -nr;
   i=0;
   do {
      str[i++] = nr % 10 + '0';
   } while ((nr /= 10) > 0);
   if (sign < 0)
      str[i++] = '-';
   str[i] = '\0';
   for (i=0, j=strlen(str)-1; i<j; i++, j--)
   {
      c = str[i];
      str[i]=str[j];
      str[j] = c;
   }

   return(0);
}
```

Please write to:
Laura M. Morrison
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Programming the Amiga in Assembly Language:

In this article I'll talk about double precision floating-point numbers, use them to draw the Julia set, and start you off with some elementary menus. There's also a little surprise that will let you scroll within the Julia set. All this is in preparation for the next article when we'll use a faster method for depicting the Mandelbrot/Julia sets, use full fledged menus and a zoom routine. First things first, though.

DOUBLE PRECISION

A few articles ago in Part II of this series (April 1992) I talked about single precision numbers and we worked a little with them. But their accuracy is limited, and many programs such as drawing fractals require much more precision. Double precision (DP) numbers use 64 bits to express a number. The left-most bit (63) is the sign (0=positive, 1=negative), the next 11 bits are the exponent, and the remaining 52 bits are the mantissa. This is the reverse order of single precision numbers.

The number to be converted to DP must first be written as a power of two. Ten, for example, is 1010; now make this a number less than one by moving the decimal four places to the left, or .1010*2^4. In DP however, since the first number on the left must always be a 1, it is assumed to be there, is ignored and dropped. So 10 is actually written as .010*2^3. The exponent is derived by adding $3FF to the original exponent - in this case 3; so the full exponent is $402.

The mantissa is derived be ignoring the decimal and writing the number as a 52 bit number adding zeros if necessary. Our mantissa in this example is 01000000... or $40... and the full number is then ($40)(24)(00)(00)(00)(00)(00)(00). If this had been a negative number, add $80 to the exponent. The only exception to this procedure is 0 - it's always 64 zeros.

Let's try one more example of 2.5. This is written as 10.1 and normalized to .101*2^2 and in DP as .01*2^1. The exponent is $3FF+1 or $400, the mantissa is 01000000... or $40... The entire number then is $4004000000000000.

Unfortunately, to get this accuracy we must sacrifice something and that's speed. Since each DP number is 64 bits it takes two registers to hold one number. Most basic math procedures involve two numbers so it takes four registers, for example, just to add. The library that handles all basic DP math is mathieeedoubbas.library. It has the same offsets for it's functions as the mathffp.library (see Table I). Most of it's routines require that you put the first number in registers d0,d1 and the second number in registers d2/d3.

DPMATHMACROS.I

The DP routines are in Listing 1, DPMATHMACROS.i; they follow the general procedures of SPMATHMACROS.i from Part II. Most of the macros are longer, however, since you might need to include two parameters for adding, subtracting, etc. The second half of a number is always stored one long word after the first half, so it's location can always be referenced by adding "+4" after the parameter. For example, if the value in AA is the first parameter passed the second half of AA is in "\1+4".

The macros for sign, cosine, square root, etc. have the same offsets as those in the mathtrans.library. Again, most macros allow you to assume or define the second half of the label you're using. These DP functions are in the mathieeedoubtrans.library. Both libraries and the macro file are included on the magazine disk.

Since many of the DP math operations involve moving double long words I tried to write the MOVEDP macro to handle the most common cases - data registers to data registers, labels to labels, labels to data registers, and d0/d1 to a label. Notice the extensive use of the command MEXIT to quit the macro.

Now we'll use DP numbers to explore the Julia set. Named for French mathematician Gaston Julia, the Julia set reiterates a function over and over again until it either remains stable or explodes out to infinity. The length of time it takes to do this is color coded and is the most interesting part of the display.

The longest routine in the program, CONVERTDP, will convert an ASCII string number to it's DP equivalent. This routine is similar to the one in SPMATHMACROS.i, but in case you've forgotten how it works, I'll review the DP convert routine. The general procedure is to -

1) determine the sign value and save it as 0 or #$80
2) convert the number to a whole number times 10 to a power
3) use DPFLT to convert this number to a DP number
4) divide the DP number by 10 to the power

For example, .12345 is 12345*10^5 so 12345 would be converted to DP format. Once in d0/d1 it would be divided by 10 five times. The routine is heavily commented so try following through it with your own example. The two major considerations are to watch where the decimal is and don't try to convert a 0 - just make it 64 zeros.

ASCII TO DOUBLE PRECISION

In the Julia set all of our numbers will be between -2 and +2 so we'll have, at the most, only one whole number. The rest of the digits can go to precision in the decimal - I allowed for up to 15 spaces after the decimal. Since the routine to convert a string number to DP is so long and only used at the beginning of the program, I made it a subroutine instead of a macro. There is nothing wrong however with making it a macro, but it will be repeated everywhere in the program where it's called. In this routine register assignments are:

```
d4 = sign              d7 = # of digits right of decimal
a0 = decimal string    a2 = decimal flag
```

Part V - Dinner With Julia

by William P. Nee

Register a0 contains the location of the decimal string. If the first character value there is a minus sign then bit 31 of d4 is set. The phrase #'-' means to look for the ASCII value for that character; not all of us can remember these values, but the assembler knows what we want. Then check to see if there is a decimal; if there is, flag a2 and clear d7. Now each successive character is checked to see if it's between 0 and 9. If it is, the contents of d0/d1 are multiplied by 10 and the current digit added. The registers are multiplied by 10 using shifts. One left shift multiplies by two and the result is stored in d2/d3; three more left shifts multiplies by eight and then d0/d1 and d2/d3 are added giving the same result as multiplying by 10. The ROX command means to rotate and include the carry value from the previous ASL. This conveniently shifts double registers. Register d7 keeps track of the number of characters to the right of the decimal.

When the entire string has been checked, first see if it was just a 0. If so, we're finished since DP 0 is 0. Assuming that we have a value in d0/d1 however, how do we convert that to a DP number? The routine DPFLT will only convert the number in d0 and we've probably got values in both d0 and d1 for precision.

The maximum exponent in this convert routine is #$43F, so put that in d6. Keep doubling the value in d0/d1 with left shifts and keep decreasing the exponent by subtracting 1. When there is finally a carry (d0 has rotated a value outside it's register) we have the number's exponent in d6 and mantissa in d0/d1. This corresponds to dropping that understood '1' I talked about earlier. The mantissa, however, is all the way to the left so we need to move it 12 bits to the right to make room for the exponent and sign. The exponent is in the lower half of d6, so a SWAP will reverse the high and low words moving the exponent to the left word; four left shifts will then move the exponent to the far left of d6 where it can be combined with d0 to make the complete DP number.

If a2 is clear then there is no decimal portion to worry about. If it isn't, however, keep dividing d0/d1 by 10 and keep decreasing d7 until d7 reaches zero. The only procedure left to do is add the sign value in d4 to d0. Now d0/d1 contains a signed DP number corresponding to the original decimal string.

TABLE I

MATHIEEEDOUBBAS FUNCTIONS

NAME		OFFSET
DPFIX	-30	converts dp# in d0/d1 to a whole # in d0
DPFLT	-36	converts whole # in d0 to a dp# in d0/d1
DPCMP	-42	compares dp#s in d0/d1 and d2/d3
DPTST	-48	tests the dp# in d0/d1
DPABS	-54	makes the dp# in d0/d1 always positive
DPNEG	-60	multiplies the dp# in d0/d1 by -1
DPADD	-66	adds the dp#s in d0/d1 and d2/d3; result in d0/d1
DPSUB	-72	subtracts the dp# in d2/d3 from d0/d1
DPMUL	-78	multiplies the dp# in d0/d1 times d2/d3
DPDIV	-84	divides the dp# in d0/d1 by d2/d3

MATHIEEEDOUBTRANS FUNCTIONS

DPATAN	-30	arctangent of dp# in d0/d1
DPSIN	-36	sine of dp# in d0/d1
DPCOS	-42	cosine of dp# in d0/d1
DPTAN	-48	tangent of dp# in d0/d1
DPSINCOS	-54	returns sine in d0/d1, cosine pointer in a0
DPSINH	-60	hyperbolic sine in d0/d1
DPCOSH	-66	hyperbolic cosine in d0/d1
DPTANH	-72	hyperbolic tangent in d0/d1
DPEXP	-78	exponent of d0/d1 (E^(d0/d1))
DPLOG	-84	natural log of d0/d1
DPPOW	-90	dp# in d2/d3 to the power in d0/d1
DPSQRT	-96	square root of d0/d1
DPTIEEE	-102	convert sp# to dp#
DPFIEEE	-108	convert dp# to sp#
DPASIN	-114	compute arc sine of dp# in d0/d1
DPACOS	-120	compute arc cosine of dp# in d0/d1
DPLOG10	-128	compute log, base 10, of dp# in d0/d1

COMPLEX NUMBERS

Earlier I said that Julia sets were a representation of repeated iterations of a function. Since a function with just one unknown would produce a straight line we'll use a function with two unknowns; the second unknown, however, will represent an imaginary number or i (the square root of -1). So any number, call it Z, is really X+iY. The function we'll keep repeating is Z^2, or, in other words, new Z=Z^2. The new real part of Z is X^2-Y^2 and the new imaginary part is 2*X*Y. To compute the Julia Set we'll always add a constant value to our result - another complex number C composed of a real A and imaginary B (or JA+iJB). Now to square our number - NEWX = X*X-Y*Y+JA, and NEWY = 2*X*Y+JB

Anytime the distance of our new point is more than 2 units long, that value will eventually explode. An easy way of expressing this is "IF X*X+Y*Y>4" avoiding square root computation. But how long do you keep iterating before you can assume a value will never reach that condition? That depends on your time and patience. A low iteration count (around 50) can produce satisfactory results for low-scale pictures. It may take an iteration count of 1000 for very high-scale pictures.

All of the Julia Set is within a box -2 to +2 in both the X and Y directions (the Z value); the constant C you pick can be anywhere inside that box (a +/-JB will produce the same result). If you want to enlarge one portion of the display, change the X and Y values but keep C the same. To look at just the upper one-quarter, X could go from -2 to 0 and Y could go from 0 to 2. The scale for X and Y is the distance for each divided by the size of the display. Points that fall within the Julia Set are all colored the same (usually black); all of the rest are colored by establishing a color relationship to the iteration count. The entire process can be computed as:

```
XLEFT=-2:XRIGHT=+2:XSCALE=(XRIGHT-XLEFT)/64
YBOTTOM=-2:YTOP=+2:YSCALE=(YTOP-YBOTTOM)/48
FOR I=0 TO [illegible]:X=XLEFT+I*XSCALE
FOR J=0 TO [illegible]:Y=YBOTTOM+J*YSCALE
A=X:B=Y:FOR C=0 TO ITERATION
ASQR=A*A:BSQR=B*B
IF ASQR+BSQR>4 [illegible] GOTO LOOP
B=2*A*B+JB
A=ASQR-BSQR+JA
NEXT C:PSET(I,J),[illegible]
[illegible]NEXT J,I
```

LISTING 2

This is the same procedure as Listing 2, the machine language program. I've put several coordinates you might want to try at the end of the listing. They all cover the entire Julia set so you may want to enlarge them by changing the X and/or Y distance; be sure to leave the last two coordinates the same or you'll get a different Julia set.

Since there will always be six coordinates to use, the "convert_to_DP" routine must be able to read all six of them. Any mistake will cause d6 to come back with an error flag and will terminate the program. This probably means a value has too many digits. Space is reserved for the six variables using DC.L 0 at the end of the listing.

The most used portion of the program is the inner loop where the iterations are carried out. After computing Asquare and Bsquare the squares are added and compared to 4; if the value is lower the program computes a new real and imaginary value adding JA to the real portion and JB to the imaginary portion.

TABLE II

MENU STRUCTURE (30 bytes)

0	long	pointer to next menu structure or 0
4	word	left edge of menu - relative to screen+border
6	word	top edge of menu - set to 0
8	word	width of menu
10	word	height of menu - set to 0
12	word	flags - menu enabled ($1)
14	long	pointer to menu name, null-terminated
18	long	pointer to first menu item structure
22-28	bytes	used by Intuition - set to 0

MENU ITEM STRUCTURE (34 bytes)

0	long	pointer to next menu item structure or 0
4	word	left edge of item, relative to menu
6	word	top edge - use 0 for top item
8	word	width of menu item
10	word	height of item box
12	word	flags - CHECKIT ($1) ITEMTEXT ($2) COMMSEQ ($4) MENUTOGGLE ($8) ITEMENABLED ($10) HIGHIMAGE ($0) HIGHCOMP ($40) HIGHBOX ($80) HIGHNONE ($C0) CHECKED ($100)
14	long	mutualexclude
18	long	item fill - pointer to image or text
22	long	select fill - pointer to image or text
26	byte	command key
27	byte	padding - set to 0
28	long	pointer to sub-item else 0
32	word	next select - set to 0

If the sum of the squares is 4 or more the point is outside the Julia set and colored accordingly. I've included four different color routines you could use based on the iteration count -

```
1) COUNT AND #31
2) COUNT AND #15
3) (COUNT AND #15) + #16
4) COUNT/#64
```

Another method of coloring is to assign a color value to each possible count; this is called a "hash" table. If the count was 973, you would just get the 973rd. value from your color table and use it.

Any point that never reaches 4 by the end of the iteration count is probably within the Julia set and is colored with whatever color you decide to use. Most programs use black but Amiga Palette 17 is also a nice choice. There are even ways to color points within the set, but I'll leave that up to you to figure out. I've added a new palette that you can use by passing it's name with the PALETTE macro.

There are several DP numbers declared as variables near the end of the program. Although only a few of them are used I've included them as examples if you want to see how good you are at converting numbers. All the "numberstring" values are the locations of some Julia Sets you might want to try. If you add your own be sure to enclose the string with a single quote and follow it with a comma and a zero (NULL TERMINATED). When you've typed in the program save it as JULIA.ASM. Then assemble A68K JULIA.ASM and Blink JULIA.O. Run this program from CLI as JULIA; re-assemble and Blink whenever you change coordinates or palettes.

SCROLLING

I said that I have a little surprise for you. You can make your screen and window bigger than 320 by 200, draw off the screen, save the picture and then scroll through it. Sort of like a super-bitmap, but without all the hassle. Here's how I made a lo-res, non-interlaced, 32 color 640X400 picture. Change the coordinates of the Julia set so only the bottom one-fourth will show on the screen; I used -1.4, 0, -1.1, 0, -.194, .6557. Then change the screen and window dimensions to 640X400. For coloring, I used a count of 512 and ANDed each count with 31.

Keep the scale the same, but increase the across distance to 640 and the down distance to 400; the rest of the picture will be drawn off the screen. When it's finished you need to save it. I like to use PictSaver since it runs in the background and you can save a picture at any time (CTRL/LALT/P). Once you've saved your picture, show it with a program like SuperView. With SuperView you do need to use the -m option since the program thinks that the larger screen is a high-resolution picture. I used SUPERVIEW -M -X1 -W320 -H200 JULIADEMO.PIX to show my picture; then you can hold down the left mouse button and scroll around the picture. For your convenience, the Public Domain programs PictSaver and SuperView along with their docs and JuliaDemo.Pix are on the magazine disk.

MENUS

No dinner is complete without a good menu from which to choose. So let's talk a little about menus - subtle, eh! Just as screens and windows have a predefined structure, so do menus. Table II lists the elements of the menu structure. But a menu title is usually meaningless without some items under it. These items have their own structure called MenuItem and again are listed in Table II. In turn, menu items may each have their own sub-menu items; fortunately their format is the same as menu items. Finally, there must be something to read in each menu. Intuition can print null-terminated string text in a menu, gadget, alert, etc. As you would suspect, this text must be in a structured format; see Table III for the IntuiText structure.

TABLE III

INTUIMESSAGE STRUCTURE (52 bytes)

0		IM.MESSAGE
20	long	IM.CLASS - IDCMP flags
24	word	IM.CODE - menu/item/sub-item selection
26	word	IM.QUALIFIER - keyboard information
28	long	IM.IADDRESS - address of Intuition object
32	word	IM.MOUSEX - mouse X coordinate
34	word	IM.MOUSEY - mouse Y coordinate
36	long	IM.SECONDS - current time
40	long	IM.MICROS - current time
44	long	IM.IDCMPWINDOW - address of window of this IDCMP
48	long	IM.SPECIALLINK - used by system

INTUITEXT STRUCTURE (20 bytes)

0	byte	foreground pen color - text color
1	byte	background pen color - background if JAM2
2	byte	drawing mode (JAM1, JAM2, etc.)
3	byte	padding - set to 0
4	word	left edge - relative to container
6	word	top edge - relative to container
8	long	font pointer or 0
12	long	pointer to null terminated text
16	long	next intuitext - set to 0

Let's discuss the various flags in the menu structure in more detail. The only menu flag is ENABLED ($1); this means the menu will be fully visible and ready for use. The flags for a menu item are more varied to show greater variety. The available flags are:

- CHECKIT ($1) - an attribute item; can be mutually excluded
- ITEMTEXT ($2) - item will contain text
- COMMSEQ ($4) - there will be a command key
- MENUTOGGLE ($8) - toggle the checkmark
- ITEMENABLED ($10) - menu item is initially enabled
- HIGHIMAGE ($0) - display user's image
- HIGHCOMP ($40) - highlight by complementing colors
- HIGHBOX ($80) - draw a box around the selected item
- HIGHNONE ($C0) - no highlighting
- CHECKED ($100) - this item will be checked

Another field that needs explaining is mutualexclude. If you've flagged a menu item with CHECKIT then it can be used while all other options are excluded. Since you want an attribute item (as opposed to an action item) to only be selected once, you tell Intuition to exclude all other attribute items with a 32-bit long word; the 1's in this word represent the other menu items to be excluded when this menu item is picked. A value of $FFFFFFFE would exclude all other menu items when the first one was selected; a value of $1 would always exclude the first item when any other item was chosen.

The command field allows you to access a menu item from the keyboard by pressing right Amiga plus a key. The ASCII value for this key must appear in this field. Intuition will add 'A' and the character after the item name, so be sure to leave room for it. In the same manner, checkmarks are added before the attribute item name so also leave room for them.

ATTACHING MENUS

Now, how do we attach our menus to a window? The Intuition command SetMenuStrip will attach all menus, items, sub-items, and text to your window. Put the window address in register a0 and the menu address in a1. Use the command ClearMenuStrip to remove all menus by putting just the window address in a0; call this function before closing any window with a menu.

Before we can read which menu, item, or sub-item the user has selected we must make sure our window has the proper IDCMP flags (see Part II) needed for menus. MENUPICK will let you know if the user has released the RMB (right mouse button) over a menu; MOUSEBUTTONS will keep track of mouseX and mouseY coordinates.

The general procedure to check for a message is to use WaitPort to identify the UserPort, use GetMsg to see if the user has activated a menu, clear window gadget, etc., sort the message into one of three categories, and use ReplyMsg to immediately answer the message (see Table III). The three types of messages are:

```
IM.CLASS - the IDCMP flags
IM.CODE - menu values
IM.QUALIFIER - raw key values
```

If the IM.CLASS value is the same as the IDCMP flag for closing a window, then go to the program routine that closes that window. If it's the MENUPICK value then a menu has been chosen and we need to look at the menunumber value in IM.CODE to see which menu item and sub-item were selected.

The menunumber is a 16-bit value; the right-most five bits (4-0) are the menu numbers starting with menu0; the next six bits (10-5) are the item numbers starting with item0; the last five bits (15-11) are the sub-items, again starting with sub-item0. By using shifts and rotates you can mask these three locations and separate their values. A menunumber value of $FFFF means that no menu item or sub-item has been selected; this is called MENUNULL.

MENU.i

In Listing 3 (MENU.i) I've included most of the menu values you need. I'll talk about some more features of menus in the next article, but this is enough to get you started. The three macros open a menu strip, evaluate the menunumber, and close a menu strip.

To use all of this information I wrote a short program (Listing 4) that uses the WBench screen and sets up a menu with two items. Both items are attribute items with the first one checked, and both have command keys. Select an item using the mouse button or Amiga M or Amiga I; after selecting an item a short message appears for a few seconds. Close the window using the close gadget or by pressing ALT/LAMIGA. Try using different combinations of menu item flags and different pen colors.

To write the two menu messages, I used a new routine - Text. This is a graphics command that will print text at any location in any color. The LOCATE macro positions the pens. I added a GTEXT macro that requires the color, address of the GTEXT, and it's length. The GTEXT is not null-terminated. I've also added a DELAY macro to DOSMACROS.i; it uses the passed value to call the DOS function Delay. Copy or assemble this program as MENU.ASM. Listing 4, MENU.i, DOSMACROS.i, and GFXMACROS.i are included on this disk.

NEXT TIME

You can see that as you increase the number of menus, items, and sub-items the source code gets unwieldy very quickly. Try to develope your own macros to handle the repetitive portion of menu creating; next time I'll show you how I did it. In the next article we'll also use scaled numbers to draw the Mandelbrot set, string gadgets to input coordinates, and include a zoom routine to enlarge your drawings.

Listing One

```
[illegible]
```

```
    endc
    endc
    ifc     '\1','d1'       ;d0,register
    ifc     '\2','d0'
    move.l  d2,d0
    move.l  d1,d1
    mexit
    endc
    ifc     '\1','d2'
    move.l  d2,d4
    move.l  d3,d5
    mexit
    endc
    ifc     '\2','d6'
    move.l  d2,d6
    move.l  d3,d7
    mexit
    endc
    endc
    ifc     '\1','d4'       ;d4,register
    ifc     '\2','d0'
    move.l  d4,d0
    move.l  d5,d1
    mexit
    endc
    ifc     '\2','d2'
    move.l  d4,d2
    move.l  d5,d3
    mexit
    endc
    ifc     '\2','d6'
    move.l  d4,d6
    move.l  d5,d7
    mexit
    endc
    endc
    ifc     '\1','d6'       ;d6,register
    ifc     '\2','d0'
    move.l  d6,d0
    move.l  d7,d1
    mexit
    endc
    ifc     '\2','d2'
    move.l  d6,d2
    move.l  d7,d3
    mexit
    endc
    ifc     '\2','d4'
    move.l  d6,d4
    move.l  d7,d5
    mexit
    endc
    endc
    ifeq    NARG-2
    ifc     '\2','d0'       ;label,register
    move.l  \1,d0
    move.l  \1+4,d1
    mexit
    endc
    ifc     '\2','d2'
    move.l  \1,d2
    move.l  \1+4,d3
    mexit
    endc
    ifc     '\2','d4'
    move.l  \1,d4
    move.l  \1+4,d5
    mexit
    endc
    ifc     '\2','d6'
    move.l  \1,d6
    move.l  \1+4,d7
    endc
    move.l  \1,\2           ;label1,label2
    move.l  \1+4,\2+4
    endc
    ifeq    NARG-1          ;label
    move.l  d0,\1           ;d0 -> label
    move.l  d1,\1+4         ;d1 -> label+4
    endc
    endm
dmove1 macro
    move.l  \1,d0
    move.l  \1+4,d1
    move.l  \2,d2
    move.l  \2+4,d3
    endm
dmove2 macro                ;label
    move.l  \1,d2
    move.l  \1+4,d3
```

[illegible]

Listing Two

```
   closelib dpmath
close_gfx
   closelib gfx
close_int
   closelib int
done
   move.l   stack,sp
   rts

   evenpc

stack      dc.l  0            ;reserve storage locations
gfxbase    dc.l  0
intbase    dc.l  0
dpmathbase dc.l  0
                              ;library names
gfx        dc.b  'graphics.library',0
   evenpc
int        dc.b  'intuition.library',0
   evenpc
dpmath     dc.b  'mathieeedoubbas.library',0
   evenpc

myscreen:
   dc.w 0,0,320,200,depth       ;depth is 5
   dc.b 0,1
   dc.w 0
   dc.w customscreen
   dc.l 0,0,0,0

mywindow
   dc.w 0,0,320,200
   dc.b 0,1
   dc.l $8
   dc.l $11800
   dc.l 0,0
   dc.l 0
   dc.l 0,0
   dc.w 0,0,0,0
   dc.w customscreen

   evenpc
across    dc.l  0
down      dc.l  0
xleft     dc.l  0,0              ;reserve space for both parts
xright    dc.l  0,0
xa        dc.l  0,0
xinc      dc.l  0,0
ybottom   dc.l  0,0
ytop      dc.l  0,0
yinc      dc.l  0,0
juliaa    dc.l  0,0
juliab    dc.l  0,0
asqr      dc.l  0,0
bsqr      dc.l  0,0
aa        dc.l  0,0
bb        dc.l  0,0
count     dc.l  0
onefp     dc.l  $3ff00000      ;first half only
twofp     dc.l  $40000000
tenfp     dc.l  $40240000
twentyfp dc.l  $40340000
one_hundredfp  dc.l  $40590000
one_thousandfp dc.l  $408f4000
   evenpc
colormap                     ;my new palette
  dc.w $000,$f0f,$d0f,$b0f,$90f,$70f,$50f,$40f
  dc.w $30f,$00f,$01d,$05b,$079,$097,$0b5,$0d3
  dc.w $0f0,$3f0,$5f0,$9f0,$bf0,$df0,$ff0,$fe0
  dc.w $fd0,$fc0,$fa0,$f80,$f60,$f40,$f20,$f00
   evenpc
numberstring                 ;try me
   dc.b  '-1.4,1.4,-1.4,1.4,-.194,.6557',0  ;count=128
  ;dc.b  '-1.4,0,0,1.3,-.194,.6557',0 ;ZOOM count=512
   ;dc.b '-1.5,1.5,-1.4,1.4,0,1',0
   ;dc.b '-1.2,1.2,-1.3,1.3,.27334,.00742',0
   ;dc.b '-1.8,1.8,-1.8,1.8,.360284,.100376',0
   ;dc.b '-.891858,-.0147619,.277544,1.16576,.360284,.100376',0  ;ZOOM
   ;dc.b '-.2,.2,-.15,.15,-1.5,0',0  ;ZOOM
   evenpc
 end
```

Listing Three

```
;Listing 3
;MENU.i
menunull        =  $1F
menuenabled     =  $2000

lowcommwidth    =  $20
commwidth       =  $1B
lowcheckwidth   =  $20
checkwidth      =  $13

menuenabled     =  $1
checkit         =  $1
itemtext        =  $2
commseq         =  $4
menutoggle      =  $8
itemenabled     =  $10
highimage       =  $0
highcomp        =  $40
highbox         =  $80
highnone        =  $C0
checked         =  $100

nomenu          =  $1F
noitem          =  $3F
nosub           =  $1F
menunull        =  $FFFF

im_class        =  $14
im_code         =  $18
im_qualifier    =  $1A
im_iaddress     =  $1C
im_mousex       =  $20
im_mousey       =  $22
im_seconds      =  $24
im_micros       =  $28
im_idcmpwindow  =  $2C
im_speciallink  =  $30

;MACROS
openmenu    macro    ;menu
   lea       \1,a1
   movea.l   window,a0
   intlib    setmenustrip
   endm
closemenu   macro
   movea.l   window,a0
   intlib    clearmenustrip
   endm

full_menunumber macro
   moveq     #0,d0
   moveq     #0,d1
   moveq     #0,d2
   move.w    d3,d0
   move.w    d3,d1
   move.w    d3,d2
   andi.w    #31,d0    ; = menu #
   lsr.w     #5,d1
   andi.w    #63,d1    ; = item #
   rol.w     #5,d2
   andi.w    #31,d2    ; = subitem #
   endm
```

Listing Four

```
;Listing 4
;MENU.asm
   include   execmacros.i
   include   intmacros.i
   include   gfxmacros.i
   include   dosmacros.i
   include   menu.i

start
```

The complete set of listings for Programming the Amiga in Assembly Language—Part V can be found on the AC's TECH disk.

Please write to:
William P. Nee
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722

Trading Commodities:

by Scott Palmateer

Introduction - The Story of a Mouse with Three Buttons

I have a three-button mouse. Most Amiga applications support only two-button mice. I used to wonder what the middle button was for, since although the Amiga's operating system has the capability to recognize the middle button, nothing happens when it is pressed. Then I read about some public domain (PD) programs that cause the Amiga to pretend that the shift key is being pressed when the user presses the middle button. "Ah," I thought, "finally a use for my fancy three-button mouse." Some programs, including *Imagine*, the *Workbench*, and undoubtedly others, allow the user to do "multi-selects"; that is, if the user holds a shift key down while selecting items on the screen with the left button, all the items, not just one, become selected. Masking the middle button press as a shift key press frees the user from having to do this multi-selecting with both hands. This is obviously very convenient.

At the same time, programs such as *ProWrite* have other "key-button" combinations. For example, in ProWrite if one holds down the alt key while using the menus to open a file, the file requester will display all files for a given directory, not just ProWrite files. I have always found this a bit awkward, and so I have always wished for a way to let my middle button pose as the alt key.

I could easily obtain the PD program to convert my middle button into a shift key, but I could not find one to make it into an alt key. Besides, even if I found one, there is no telling whether the two would work together. So, I decided to write my own program that would let me control what the middle mouse button did from one central interface, guaranteed to work with itself! That is where Workbench 2.0's "Commodities" comes in, and the result of my effort is MMB_CX, a small and simple-to-use commodity that allows the user to convert the middle mouse button into a left shift key, a right shift key, a left alt key, a right alt key, a left Amiga key, a right Amiga key, or the control key. MMB_CX also provides a "hotkey" that brings up a window through which the user can control the program. By default, this "hotkey" is activated by holding down the middle button and pressing the help key, but the user can change this to any other combination.

A Bit About Input Events on the Amiga

Normally, when the user causes some kind of input (a keypress, a mouse move, etc.), a program called the input.device creates an "input event" that is introduced to the "input stream" (Figure 1). The input event is a data structure that contains information as to what kind of event it is—a keypress, for instance; it also includes information like the time of the event and the coordinates of the mouse at the time of the event. The input event then travels down the input stream, which is a chain of programs that are interested in input, such as *Intuition* and the console device. The input event visits each program, or handler, in order. The user can install his own "input handler" into this input stream; the input handler waits for certain events and acts upon them when it receives them. This action can be bringing up a window, blanking the screen or the mouse pointer, or just about anything else. But as I mentioned before, it can be difficult to get more than one of these handlers to work together. This is where Workbench 2.0's Commodities come in.

The Commodities Exchange System

Workbench 2.0's Commodities have nothing to do with money, even though brokers and an Exchange are involved! It is simply a name that the programmers have chosen to refer to this system. It is an object-oriented approach to standardizing the way the user modifies the way input events (key presses, button presses, mouse movements, etc.) are handled by the Amiga. And the best thing about Commodities is that they are guaranteed to work with one another. As I have said, there are numerous PD programs that modify these events, but they all do so in their own isolated way, and compatibility problems have occasionally surfaced when using them together. With Workbench 2.0, the user basically only has to inform the operating system that he would like to modify a certain event into another, or just monitor the input events for a certain "hotkey" combination, and the commodities library takes care of everything else for you! It is really quite elegant. There is even a function called "Hotkey" that creates a hotkey in one fell swoop! To do the same job without Commodities, I would have had to learn about opening the input device, interrupts, and other complex issues. In MMB_CX, I had only to tell the system to "listen" for middle button presses, and convert them into whatever qualifier key I had chosen at that particular moment, say a left shift key.

How Commodities Work

The Commodities system installs a custom handler, maintained by the system, into the input stream before Intuition's input handler (Figure 2). The fact that Commodities sees each input event before Intuition does is crucial to their operation. It lets the user install a commodity that swaps the functions of the left and right mouse buttons (for left-handed users); as far as Intuition is concerned, the user has done nothing unusual! In the example of MMB_CX, it allows our program to convert any middle mouse button press into a left-shift (or whatever) before Intuition realizes it.

Commodore supplies a control program, called Exchange, that allows the user to control the commodities installed in his system. This program shows all the commodities currently running. It can temporarily disable/enable any of the commodities, or permanently get rid of them. For those commodities that have some kind of control window to control their individual features, Exchange can instruct them to bring up their control window. Exchange can also tell the commodity to close its window (but keep running). Exchange acts like a commodity itself; once running, its control window can be brought up with the hotkey combination of "alt-help."

The user can use the ToolType environment of Workbench to control certain options for the commodity, such as priority. We will discuss this later in more detail.

Workbench 2.0

The Broker

A commodity is really a collection of "objects" (CxObj's), one of which is a "broker," connected together in a particular fashion, and referred to in data structure circles as a tree. The broker is the central reference point for the entire commodity, and it occupies the root position of the tree. The broker, upon receiving an input event from the input device, converts the event into a commodity message (CxMsg) and diverts it to its children (the CxObj's connected to the broker). When they are done doing whatever they want to with it, the broker sends the CxMsg to its next destination, which can be another broker, or the next handler in the input stream (Figure 3). Brokers are organized by priority; by setting broker A's priority higher than broker B's, you can make sure that broker A sees the input event before broker B does. This can be important when you have more than one commodity running at the same time.

The Filter Object

There are a variety of types of CxObj's. One of the most important, because it is used in almost every conceivable commodity, is the Filter CxObj. Its function is to filter out any unwanted input events before passing them along to its children. For instance, in the case of MMB_CX, we are interested only in events that involve a middle mouse button, so we inform the Filter that it should let only those events pass by to its children; if the event does not meet the Filter's standards, it is passed to the Filter's siblings (if the Filter has no siblings, it is passed back to the parent). This prevents our commodity from having to do too much work. All CxObj's, including Brokers, pass CxMsg's in this pattern: first to any children, then to any siblings, and finally to the parent.

I should point out a few things before I explain anything else. All of these CxObj's are implemented and maintained by the system. I do not have to know exactly how they work; I have to know only that they do indeed work. This is what I meant when I said earlier that the Commodities system is an object-oriented system. Constructing a commodity involves little more than assembling the appropriate objects in the correct way; the details of their actual operation are unimportant. There are enough types of objects to let you do just about anything you want.

The Sender Object

Another type of object is the Sender CxObj. Its job is to send my commodity a message when it receives a CxMsg from its parent. A typical use for the Sender CxObj is to have it connected to a Filter CxObj; when the Filter encounters an input event that it likes, it passes it down to the Sender. The Sender, upon receiving the message, alerts my commodity. My commodity can then take some appropriate action. This is how hotkeys are implemented. In the case of MMB_CX, a Filter looks for an "middle mouse button - help key" combination; when it finds one, it passes the message to a Sender, which sends yet another message to MMB_CX, which pops open the command window.

The Sender CxObj sends my commodity a CxMsg containing a COPY of the input event that it received, along with an application-specific code number, supplied by me. The fact that the input event is a copy is important. It means that the Sender does no modification to the input event, and any fiddling we do with it in our commodity

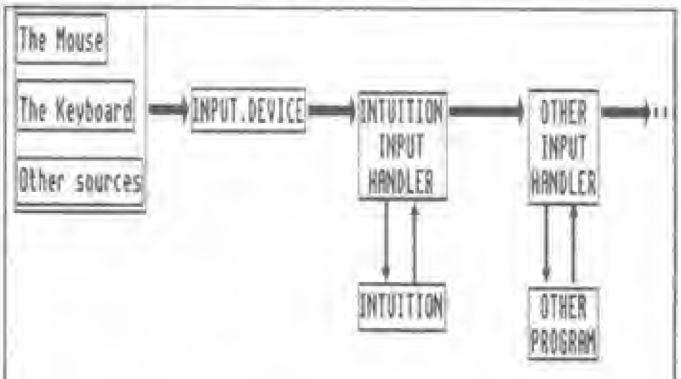


Figure 1: The normal Input Stream
(extracted from the Amiga ROM Kernel Reference Manual: Libraries)

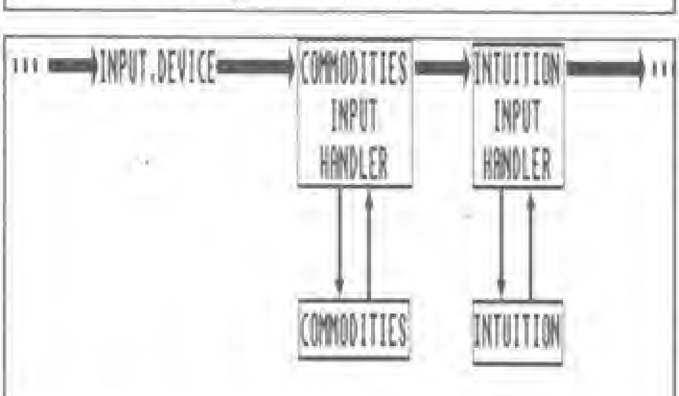


Figure 2: The Input Stream when Commodities are running
(extracted from the Amiga ROM Kernel Reference Maunal: Libraries)

application will not affect the input event at all. If we want, there are objects that can change the input event.

After the Sender has sent its application a message, it follows the typical CxMsg passing pattern. When using Commodities to implement a hotkey, the Sender usually has a Translator CxObj for a sibling.

The Translator Object

The Translator CxObj converts one input event into another, or into a string of them. The Translator can even "swallow" the input; that is, it can cause the system to pretend that the input event never occurred. This is what usually follows a Sender in a hotkey application. Once the commodity is informed that we have activated our hotkey combination, we do not want the input event to continue down the input stream. Using a Translator CxObj, we simply get rid of it.

The Translator can also be used to transform an event into a string of them. You might use this feature to convert a function key event into a string of characters, or anything else that you find useful.

We cannot use a Translator in MMB_CX, though, because a Translator has to be told ahead of time what to use to replace a particular input event. MMB_CX allows the user to change a middle mouse button event into one of several others. Another type of CxObj, the Custom CxObj, is appropriate here.

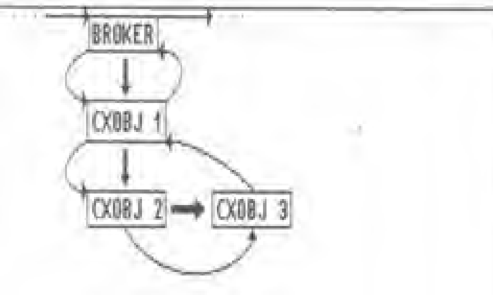


Figure 3: The path of an input event/CxMsg through a Commodity (denoted by the thin curved line). Note that Filter objects only pass the CxMsg to their children if the CxMsg matches the Filter's criteria. Also, come objects can be used to completely remove input events from the stream.

The Custom Object

The Custom CxObj is similar to the Sender CxObj in that it informs my commodity upon receiving a CxMsg. However, the Custom CxObj does not send a CxMsg with a copy of the input event; it calls a function, defined by my commodity, with a pointer to the *actual* input event. Since the function called by the Custom object works with an actual input event, it must be extremely careful, and extremely quick. It must also be declared in a special way. Using SAS/C, the function should have the __saveds keyword preceding the function definition. This keyword saves the "global data segment pointer." This is a technical point which I will not go into here; suffice it to say that our Custom CxObj's function will be called not by our commodity, but indirectly by input.device. It therefore needs to run a little differently from our other functions.

In MMB_CX, the function requalify() is called by a Custom CxObj. Requalify() simply takes the middle mouse button qualifier out of the input event and replaces it with whatever we have chosen to have the middle button signify, leaving everything else intact. It then returns control to Commodities.

There are other CxObj types, but MMB_CX does not use them, so I will not discuss them here.

How MMB_CX Is Organized

Constructing our commodity begins with creating the Broker. This is done with the following statement:

```
        broker   =   CxBroker(&mynewbroker,
&errorvariable);
```

where "mynewbroker" is an instance of an initialized NewBroker structure. The NewBroker structure has various fields to tell Commodities how to use this commodity, such as the name of the commodity, whether it has a popup control window, etc. The variable "errorvariable" is filled in by the CxBroker() function if there is some error creating the Broker. The error could be lack of memory, or the fact that another commodity with the same name is already running. Our Broker has two immediate children, both of them Filter objects. The first Filter object is used to "listen" for our hotkey event; the second is used to listen for middle mouse button events. This order is important, because remember that our function requalify() removes the middle mouse button qualifier from the event and replaces it with something else. If we removed the middle mouse button qualifier before our hotkey Filter got to examine the input event, it would never acknowledge that our hotkey had taken place!

The hotkey Filter itself has two children, a Sender and a Translator. The Sender notifies MMB_CX that the hotkey combination has been detected, and the Translator removes the event from the input stream. This structure, a Filter with a Sender and Translator as children, is created in one statement:

```
        filter = hotkey(hotkeydescription, port, id);
```

where "hotkeydescription" is a description of the hotkey combination we are looking for, "port" is a pointer to a message port that we use for communication with Commodities, and "id" is a special code the Sender places in the CxMsg so that we know why we are receiving the message. The three objects could be constructed individually, but using the Hotkey() function is easier to understand at a glance.

Description Strings and the InputXpression Structure

The description for the hotkey is a normal C string consisting of certain keywords. These keywords are used to describe a large variety of input events. The string is formatted into four basic fields:

```
        [class]  [[-]qualifier|synonym]  [[-]
upstroke]  [highmap|ANSICode]
```

(the above is taken from the Amiga ROM Kernel Reference Manual: Libraries. The square brackets imply that the option contained within is optional, the curly braces indicate that the option within may be repeated as many times as desired; the vertical bar means "or".) The first field, "class", indicates the general type of input event. It can be any of the following:

```
"rawkey"
"rawmouse"
"event"
"pointerpos"
"timer"
"newprefs"
"diskinserted"
"diskremoved"
```

The class names should be fairly self-explanatory. "Rawkey" means a keyboard event, and "rawmouse" means a mouse event. If the "class" is left out of the string, "rawkey" is assumed.

The next field is the qualifier field. It further classifies and specifies the event from the class level. It can have any of the following values:

```
"lshift"
"rshift"
"capslock"
"lalt"
"ralt"
"lcommand"
"rcommand"
"numericpad"
"repeat"
"midbutton"
"rbutton"
"leftbutton"
"relativemouse"
```

Preceding the qualifier name by a dash tells Commodities to ignore that qualifier. For instance, if we wanted to wait for left-shift events, and we did not care whether the user was simultaneously holding down the left alt key, our description string would be:

```
"rawkey -lalt lshift"
```

Notice that mouse buttons are listed above, too. That is because they are valid qualifiers. We can instruct the commodity to wait for mouse button events by listing these in our description string.

The "synonym" section is used to indicate synonyms for the qualifier keys, as when the user means both shift keys (or both alt keys). The synonyms accepted are:

```
"shift"
"alt"
"caps"
```

"shift" means to accept either shift key, "alt" means to accepts either alt key, and "caps" means to accept either the caps lock key or either of the shift keys. As with qualifiers, the synonyms can accept the dash, signifying that their meaning is to be disregarded.

Next, the user can specify "upstroke", in which case only the lifting of a key will matter to the Filter. If the user specifies "-upstroke", both upstrokes and downstrokes are considered matches by the Filter.

The last field gets down to the individual key. The "highmap" can be any of the following strings:

```
"space"
"backspace"
"tab"
"enter"
"return"
"esc"
"del"
"up", "down", "right", "left"
"f1", "f2", ...., "f10"
"help"
```

So for our example commodity MMB_CX, the default description string for our hotkey is:

```
"midbutton help"
```

which tells the Filter to look for the press of the help key while the middle button is being held down.

We can also specify an "ANSICode" instead of the "highmap". The ANSICode is a particular key, derived from the current keymap. For instance, if we want to watch for the press of the

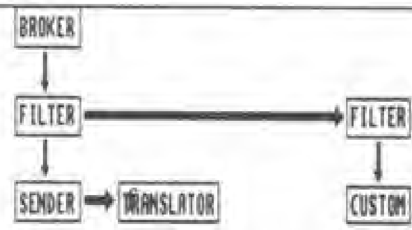


Figure 4: The structure of MMB_CX. The first filter is used for the hotkey combination, and its first child, the Sender, notifies MMB_CX when the combination is detected. The filter's second child then removes the input event from the system. The second filter is used to detect middle mouse button events. Its child, the Custom object, calls our function requalify() to convert the event.

"I" with the control key, our description string is:

```
"control a"
```

Keep in mind, that depending on the current keymap, the "A" key may or may not produce an ASCII "a".

Another way to describe input events is to use the InputXpression structure. It contains fields that correspond to a description string, and the commodities.library contains functions to convert a description string into an InputXpression structure. I use both structures in MMB_CX.

Using Filter and Custom Objects Together

Our Broker's second child is another Filter object, with a Custom object as its only child. This Filter is the one that waits for middle mouse button events. When it finds one it sends it down to its child, the Custom object, which calls our function, requalify().

List of Advertisers

Company	Pg.	RS Number
Amiga Exchange of CT	12	120
ASDG	CIV	103
Boone Technologies	13	105
Fairbrothers, Inc.	16	113
Interactive Digital Devices	17	104
Legendary Design Technologies	18	111
Memory Management	46	118
VisionSoft	54	124

MOVING?

SUBSCRIPTION PROBLEMS?

Please don't forget to let us know. If you are having a problem with your subscription or if you are planning to move, please write to:

Amazing Computing Subscription Questions
PiM Publications, Inc.
P.O. Box 2140
Fall River, MA 02722

Please remember, we cannot mail your magazine if we do not know where you are.

Please allow four to six weeks for processing

AC's TECH Disk
Volume 3, Number 1

A few notes before you dive into the disk!

- You need a working knowledge of the AmigaDOS CLI as most of the files on the AC's TECH disk are only accessible from the CLI.

- In order to fit as much information as possible on the AC's TECH Disk, we archived many of the files, using the freely redistributable archive utility 'lharc' (which is provided in the C: directory). lharc archive files have the filename extension .lzh.

  To unarchive a file *foo.lzh*, type *lharc x foo*
  For help with lharc, type *lharc ?*
  Also, files with 'lock' icons can be unarchived from the WorkBench by double-clicking the icon, and supplying a path.

We pride ourselves in the quality of our print and magnetic media publications. However, in the highly unlikely event of a faulty or damaged disk, please return the disk to *PiM Publications, Inc.* for a free replacement. Please return the disk to:

AC's TECH
Disk Replacement
P.O. Box 2140
Fall River, MA 02720-2140

Be Sure to Make a Backup!

CAUTION!

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that initiate low-level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PiM Publications, Inc., their distributors, or their retailers, will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This agreement may not apply in all geographical areas.)

Although many of the individual files and directories on the AC's TECH Disk are freely redistributable, the AC's TECH Disk itself and the collection of individual files and directories on the AC's TECH Disk are copyright ©1990, 1991 by PiM Publications, Inc. and may not be duplicated in any way. The purchaser however is encouraged to make an archival/backup copy of the AC's TECH Disk.

Also, be extremely careful when working with hardware projects. Check your work twice to avoid any damage that can happen. Also, be aware that using these projects may void the warranties of your computer equipment. PiM Publications or any of it's agents is not responsible for any damages incurred while attempting this project.

Requalify() then converts the middle mousebutton qualifier value into, say, a left-shift value. The Custom object then regains control from requalify(), and passes the modified event on. Any handler further down the input stream will then receive an input event that looks like the left-shift key was pressed, but you and I know that the middle button was really the one that was pressed!

Communicating with Commodities

As I said earlier, we have to set up a message port through which we receive CxMsg's from our commodity objects. These messages can be of two types, CXM_IEVENT and CXM_COMMAND. CXM_IEVENT messages are sent by Sender objects, and CXM_COMMAND messages are sent by the Exchange program. Once we receive a message at our port, we can determine its type by using the CxMsgType() function, and we can extract the code number using the CxMsgID() function. If we want, we can even extract the copy of the input event, using the CxMsgData() function.

If we received a CXM_COMMAND message from Exchange, it can have one of the following CxMsgID's:

```
CXCMD_APPEAR
CXCMD_DISAPPEAR
CXCMD_ENABLE
CXCMD_DISABLE
CXCMD_KILL
CXCMD_UNIQUE
```

The CXCMD_APPEAR is a request from Exchange that we should open our command window. Exchange knows whether or not our commodity has a control window, by setting the COF_SHOW_HIDE flag in our NewBroker structure. If this flag is not set, Exchange will not send this message to our commodity. Similarly, the CXCMD_DISAPPEAR command is a request from Exchange that we should close our command window. Exchange will not send this command if we have no command window.

CXCMD_ENABLE and CXCMD_DISABLE are sent by Exchange to temporarily enable and disable our commodity. CXCMD_KILL permanently stops our commodity from running.

CXCMD_UNIQUE is sent by Exchange when an attempt is made at running another commodity with the same name. We inform Exchange, through flags in the NewBroker structure, that 1) no commodity with the same name should be allowed to run, and 2) that Exchange should inform us if someone tries to start one. This allows us to pop open our command window if the need arises.

The Command Window

I used the new GadTools library to construct my command window. It is a very simple window, with only three gadgets; you should have no trouble figuring out the code. There is a cycle gadget to control which qualifier is going to be substituted instead of the middle mouse button. Clicking on this with the left button will cycle through all the choices. Holding the shift key down while clicking will cycle in the opposite direction. I also included standard button gadgets, labeled "Hide" and "Quit". The "Hide" gadget closes the command window without stopping the commodity. It is the equivalent of using the Exchange program's CXCMD_DISAPPEAR command. The "Quit" gadget causes the commodity to stop running. It is the equivalent of Exchange's CXCMD_KILL command.

I did not include a "close window" gadget in the window, because I find their use confusing when used with commodities: does it just close the window or does it kill the commodity? To resolve the dilemma, I chose to sidestep it.

The window and gadgets are font-sensitive. They utilize the user's default screen fonts. They take this into account each time the command window is opened, in case the user changed his font scheme. It would have been more memory efficient to allocate the fonts once at startup and free them once at the end of the program, but that would violate the new flexibility that Workbench 2.0 offers in terms of fonts.

Using MMB_CX

After compiling the program using the supplied makefile, make an icon for it by entering the following from a shell (assuming that the executable code is in RAM:):

```
        copy  SYS:Tools/Commodities/Blanker.info
RAM:MMB_CX.info
```

Click once on the MMB_CX icon and choose the Icons->Information option from the Workbench menu. Commodities use the ToolType environment to set program options conveniently, so you can adjust these to your liking. ToolTypes follow the general format "option=value". To specify some option, simply add it to the ToolType list. Options usually have some default value, so if you are happy with the default value, you do not have to include the option in the ToolType list, and your ToolType list can be completely empty. MMB_CX understands the following ToolTypes:

```
        CX_PRIORITY=<priority>
        CX_POPUP=<yes or no>
        CX_POPKEY=<hotkey description>
        MMB_QUAL=<substitution qualifier to use on
startup>
```

CX_PRIORITY is the priority at which that the Commodities system is to install MMB_CX. This number can range from -128 to +127. The higher the number, the higher the priority, and the sooner this MMB_CX will see input events. The default is 0, which is fine for most purposes.

Including a ToolType value of "CX_POPUP=yes" will instruct MMB_CX to open up its command window when it starts up. The default is "no".

CX_POPKEY is the option for specifying the hotkey description by which we bring up MMB_CX's command window. The default is "midbutton help".

The first three options listed are standard commodity options, and all commodities should support them. The last option, MMB_QUAL, is MMB_CX-specific. It tells the program what qualifier you want as your default qualifier, so that you do not have to open the command window and set it every time you start the program. The default is "lshift". If you want something different, say for ProWrite, add "MMB_QUAL=lalt" to your ToolType list. MMB_QUAL understands "lshift", "rshift", "lalt", "ralt", "lcommand", "rcommand", and "control".

Conclusion

The Commodities system allows one to easily modify the way that input events are interpreted on the Amiga. They are easily programmed, and do not take much memory at all. They are also easily learned; I am definitely no expert on them, or on programming the Amiga in general, but I managed to create something useful in a relatively short time. I also did not have to delve into the strange world of interrupts to accomplish this; I let the Amiga do it for me. Commodities have a wide variety of uses, such as screen blankers, mouse accelerators and mouse blankers, and programs like MMB_CX. Their object-oriented nature meshes nicely with the new object-oriented nature of Workbench 2.0, and ensures their expandability. In short, I found it extremely profitable to work in Commodities!

```
/*
** MMB_CX.c
**
** A Commodity to control a middle mouse button.
**
** Scott Palmateer, 17 July 1992
*/

#include <string.h>
#include <exec/types.h>
#include <exec/ports.h>
#include <devices/inputevent.h>
#include <intuition/intuition.h>
#include <libraries/commodities.h>
#include <workbench/workbench.h>
#include <workbench/startup.h>
#include <libraries/gadtools.h>
#include <intuition/gadgetclass.h>

#include <proto/exec.h>
#include <proto/intuition.h>
#include <proto/graphics.h>
#include <proto/commodities.h>
#include <proto/wb.h>
#include <proto/icon.h>
#include <proto/gadtools.h>

#define REQ_REL    37    /* required release of libraries
*/

#define MMB_NAME    "MMB_CX"
#define MMB_DESCR    "Transforms middle mousebutton
events"
#define MMB_TITLE  "MMB_CX v1.0"
#define DEFAULT_PRI       0
#define DEFAULT_POPKEY   "midbutton help"
#define DEFAULT_POPUP    "no"
#define DEFAULT_QUAL     "lshift"
#define MMB_HOTKEY       1

#define OFFSET           6
#define HIDE_GAD         0
#define QUIT_GAD         1
#define CYCLE_GAD        2
#define MMB_IDCMP        (CYCLEIDCMP | BUTTONIDCMP | \
                          IDCMP_REFRESHWINDOW)
#define HIDETEXT         "Hide"
#define QUITTEXT         "Quit"
#define CYCLETEXT        "Transform MMB into:"


/* FUNCTIONS DEFINED IN THIS PROGRAM: */
void __saveds          requalify(register CxMsg *cxm);
void                   openup(int argc, char *argv[]);
void                     closedown(int errnum);
void                     reqprint(char *txt);
void                     do_cxwindow(BOOL appear);

/*  Disable SAS/C CTRL-C handling  */
void            chkabort(void) {}


/*  REQUIRED DATA FOR cback.o: */
long _stack              = 2048,
     _priority           = 0,
     _BackGroundIO       = 0;
char *_procname          = MMB_NAME;


/* GLOBAL VARIABLES: */
char *v = "\0$VER: MMB_CX v1.0";

struct IntuitionBase *IntuitionBase  = NULL;
struct GfxBase          *GfxBase        = NULL;
struct Library          *CxBase         = NULL;
struct Library          *IconBase       = NULL;
struct Library          *GadToolsBase   = NULL;
struct MsgPort          *MMB_Port       = NULL;
struct Window           *MMB_Window     = NULL;
struct Gadget           *MMB_GList      = NULL;
CxObj                   *MMB_Broker     = NULL;
int                     QualIndex       = 0;
BOOL                    PopUp           = FALSE;


/* INITIALIZATION STATEMENTS: */
struct NewBroker MMB_NB = {
    NB_VERSION,                  /* nb_Version          */
    MMB_NAME,                    /* nb_Name             */
    MMB_TITLE,                   /* nb_Title            */
    MMB_DESCR,                   /* nb_Descr            */
    NBU_NOTIFY | NBU_UNIQUE,     /* nb_Unique           */
    COF_SHOW_HIDE,               /* nb_Flags            */
    0,                           /* nb_Pri              */
    NULL,                        /* nb_Port             */
    0                            /* nb_ReservedChannel  */
};

IX MMB_IX = {
     IX_VERSION,                  /* ix_version   */
     IECLASS_RAWMOUSE,            /* ix_Class     */
     0,                           /* ix_Code      */
     0,                           /* ix_CodeMask  */
     0,                           /* ix_Qualifier */
     0,                           /* ix_Qualmask  */
     0                            /* ix_QualSame  */
     };

#define NO_ERROR          100
#define INTLIB_ERROR      0
#define GFXLIB_ERROR      1
#define CXLIB_ERROR       2
#define ICONLIB_ERROR     3
#define GTLIB_ERROR       4
#define PORT_ERROR        5
#define CX_ERROR          6
#define SCREEN_ERROR      7
#define VI_ERROR          8
#define GAD_ERROR         9
#define WIN_ERROR         10
#define BROKER_ERROR      11
char *ErrMsg[] = {
   "Error opening Intuition Library!",
   "Error opening Graphics Library!",
   "Error opening Commodities Library!",
   "Error opening Icon Library!",
   "Error opening GadTools Library!",
   "Error opening port!",
   "Error opening Commodities Objects!",
   "Error opening public screen!",
```

```
    "Error getting screen's visual info:",
    "Error creating gadgets!",
    "Error opening window!",
    "Cannot start another MMB_CX!"
    };

char *QualLabels[] = {
    "Left Shift",          "Right Shift",
    "Left Alt",            "Right Alt",
    "Left Amiga",          "Right Amiga",
    "Control",             NULL
    };
char *QualNames[] = {
    "lshift",              "rshift",
    "lalt",                "ralt",
    "lcommand",            "lcommand",
    "control",             NULL
    };
UWORD QualValues[] = {
    IEQUALIFIER_LSHIFT,    IEQUALIFIER_RSHIFT,
    IEQUALIFIER_LALT,      IEQUALIFIER_RALT,
    IEQUALIFIER_LCOMMAND,  IEQUALIFIER_RCOMMAND,
    IEQUALIFIER_CONTROL
    };
USHORT QualCodes[] = {
    0x60,                  0x61,
    0x64,                  0x65,
    0x66,                  0x67,
    0x63
    };

/* ------------------------------------------------- */
void main(int argc, char *argv[])
{   struct Message       *msg;
    struct IntuiMessage  *imsg;
    struct Gadget        *gad;
    ULONG                msgid;
    ULONG                msgtype;
    BOOL                 quit = FALSE;
openup(argc, argv);
if (PopUp)  do_comwindow(TRUE);
while (!quit)  {
    if (MMB_Window)
        Wait ((1<<MMB_Port->mp_SigBit) |
              (1<<MMB_Window->UserPort->mp_SigBit));
    else
        Wait(1<<MMB_Port->mp_SigBit);
    while (msg = (struct Message *)GetMsg(MMB_Port)) {
        msgid = CxMsgID((CxMsg *)msg);
        msgtype = CxMsgType((CxMsg *)msg);
        ReplyMsg(msg);
        switch (msgtype) {
            case CXM_IEVENT:
            if (msgid==MMB_HOTKEY)  do_comwindow (TRUE);
                break;
            case CXM_COMMAND:
                switch (msgid) {
                    case CXCMD_UNIQUE:
                    case CXCMD_APPEAR:
                        do_comwindow (TRUE);
                        break;
                    case CXCMD_DISAPPEAR:
                        do_comwindow (FALSE);
                        break;
                    case CXCMD_KILL:
                        quit = TRUE;
                        break;
                    case CXCMD_DISABLE:
                        ActivateCxObj(MMB_Broker, FALSE);
                        break;
                    case CXCMD_ENABLE:
                        ActivateCxObj(MMB_Broker, TRUE);
                        break;
                }
                break;
        }
    }
    if ((!quit) && (MMB_Window))  {
        while ((MMB_Window) &&
               (imsg = GT_GetIMsg(MMB_Window->UserPort)))
{
            msgtype = imsg->Class;
            gad     = (struct Gadget *)imsg->IAddress;
            GT_ReplyIMsg(imsg);
            switch (msgtype)  {
                case IDCMP_GADGETUP:
                    switch (gad->GadgetID)  {
                        case HIDE_GAD:
                            do_comwindow (FALSE);
                            break;
                        case QUIT_GAD:
                            quit = TRUE;
                            break;
                        case CYCLE_GAD:
                            QualIndex = imsg->Code;
                            break;
                    }
                    break;
                case IDCMP_REFRESHWINDOW:
                    GT_BeginRefresh (MMB_Window);
                    GT_EndRefresh (MMB_Window, TRUE);
                    break;
            }
        }
    }
}
ActivateCxObj(MMB_Broker, FALSE);
closedown(NO_ERROR);
}
/* ------------------------------------------------- */


/* ------------------------------------------------- */
/* Here is the function that opens up all the libraries,
   ports, objects, etc.
*/
void openup(int argc, char *argv[])
{   CxObj *filter;
    CxObj *customobj;
    char  *x, **MMB_TT;
    int   n;
    BOOL  found;
/* open our libraries: */
IntuitionBase = (struct IntuitionBase *)
    OpenLibrary("intuition.library", REQ_REL);
if (!IntuitionBase)  closedown(INTLIB_ERROR);
GfxBase = (struct GfxBase *)
    OpenLibrary("graphics.library", REQ_REL);
if (!GfxBase)          closedown(GFXLIB_ERROR);
CxBase = OpenLibrary("commodities.library", REQ_REL);
if (!CxBase)           closedown(CXLIB_ERROR);
GadToolsBase = OpenLibrary("gadtools.library", REQ_REL);
if (!GadToolsBase)     closedown(GTLIB_ERROR);
```

```
/* Get user's tooltype prefs from Workbench or CLI: */
IconBase = OpenLibrary("icon.library", REQ_REL);
if (!IconBase)        closedown(ICONLIB_ERROR);
MMB_TT = (char **)ArgArrayInit(argc, argv);
MMB_NB.nb_Pri  =  ArgInt(MMB_TT,  "CX_PRIORITY",
DEFAULT_PRI);
x = ArgString(MMB_TT, "CX_POPUP", DEFAULT_POPUP);
if (stricmp(x, "yes")==0)  PopUp = TRUE;
x = ArgString(MMB_TT, "MMB_QUAL", DEFAULT_QUAL);
n=0;  found=FALSE;
while ((!found) && (QualNames[n]))  {
   if (stricmp(x, QualNames[n])==0)  {
      QualIndex = n;  found = TRUE;
      }
   else  n++;
   }
x = ArgString(MMB_TT, "CX_POPKEY", DEFAULT_POPKEY);
ArgArrayDone();
/* we're done with icon.library now */
CloseLibrary(IconBase);
/* get our port for communicating with Commodities: */
MMB_Port = CreateMsgPort();
if (!MMB_Port)        closedown(PORT_ERROR);
/* now create the commodity objects! */
/* first the broker: */
MMB_NB.nb_Port = MMB_Port;
MMB_Broker = CxBroker(&MMB_NB, NULL);
if (!MMB_Broker)  closedown(BROKER_ERROR);
/* this next function creates a filter/sender/translator
*/
filter = HotKey(x, MMB_Port, MMB_HOTKEY);
if (!filter)        closedown(C_ERROR);
AttachCxObj(MMB_Broker, filter);
if  (CxObjError(filter))  SetFilter  (filter,
DEFAULT_POPKEY);
/* now we create the actual MMB filter */
filter = CxFilter(NULL);
if (!filter)        closedown(CX_ERROR);
SetFilterIX(filter, &MMB_IX);
AttachCxObj(MMB_Broker, filter);
/* and here we link in our custom function */
customobj = CxCustom(requalify, 0);
if (!customobj)   closedown(CX_ERROR);
AttachCxObj(filter, customobj);
/* everything is ready; activate our Commodity! */
ActivateCxObj(MMB_Broker, TRUE);
}
/* ---------------------------------- */



/* ---------------------------------- */
/* This function closes down everything that we opened.
*/
void closedown(int errnum)
{  struct Message *msg;
do_comwindow (FALSE);
if (MMB_Broker)      DeleteCxObjAll(MMB_Broker);
if (MMB_Port)   {
   while (msg = GetMsg(MMB_Port)) ReplyMsg(msg);
   DeleteMsgPort(MMB_Port);
   }
if (GadToolsBase)   CloseLibrary(GadToolsBase);
if (CxBase)         CloseLibrary(CxBase);
if (GfxBase)        CloseLibrary(
                      (struct Library *)GfxBase);
if (IntuitionBase)  CloseLibrary(
                      (struct Library *)IntuitionBase);
if (errnum != NO_ERROR)   reqprint (ErrMsg[errnum]);
exit (0);
}
/* ---------------------------------- */



/* ---------------------------------- */
/*  Here  is  the  function  that  transforms  a  middle
mousebutton
   event into a left-shift event (or whatever else we
have
   chosen).  Note that this function should be very
simple
   and quick, since it will run as part of input.device!
*/
void __saveds requalify(register CxMsg *cxm)
{   register struct InputEvent *ie;
if ((ie = (struct InputEvent *)CxMsgData(cxm))  {
   if ((ie->ie_Code == IECODE_MBUTTON)  {
      ie->ie_Code = QualCodes[QualIndex];
      ie->ie_Qualifier &= ~IEQUALIFIER_MIDBUTTON;
      ie->ie_Qualifier |= QualValues[QualIndex];
      ie->ie_Class = IECLASS_RAWKEY;
      }
      else  if  ((ie->ie_Code  ==  (IECODE_MBUTTON  |
IECODE_UP_PREFIX))  {
            ie->ie_Code  =  (QualCodes[QualIndex]  |
IECODE_UP_PREFIX);
      ie->ie_Class = IECLASS_RAWKEY;
      }
   else if (ie->ie_Qualifier & IEQUALIFIER_MIDBUTTON)
{
      ie->ie_Qualifier &= ~IEQUALIFIER_MIDBUTTON;
          ie->ie_Qualifier |= QualValues[QualIndex];
          }
   }
}
/* ---------------------------------- */



/* ---------------------------------- */
/*  Just  a  little  utility  program  to  throw  some
information
   onto the screen...
*/
void reqprint (char *txt)
{    struct EasyStruct e;
e.es_StructSize = sizeof(struct EasyStruct);
e.es_Flags = NULL;
e.es_Title = MMB_NAME;
e.es_TextFormat = txt;
e.es_GadgetFormat = "OK";
EasyRequest (NULL, &e, NULL, NULL);
}
/* ---------------------------------- */



/* ---------------------------------- */
/* This is the function that handles opening the command
   window, through which we can control the behavior of
   the commodity.  If told to open the window, it makes
   sure that the window is not already open; if told to
```

```
   close the window, it makes sure that the window is
not
   already closed.
*/
void do_mmwindow(BOOL appear)
{   struct Screen    *screen;
    struct Gadget    *g;
    struct NewGadget ng;
    void             *vi;
    int              gheight;
if (appear) {
   if (!MMB_Window)  {
      if (!(screen = LockPubScreen(NULL)))
         closedown(SCREEN_ERROR);
      if (!(vi = GetVisualInfo(screen, TAG_END)))
         closedown(VI_ERROR);
      gheight=screen->Font->ta_YSize+OFFSET;
      g = CreateContext(&MMB_Glist);
      ng.ng_TextAttr   = screen->Font;
      ng.ng_VisualInfo = vi;
      ng.ng_LeftEdge   = OFFSET;
      ng.ng_TopEdge    = screen->WBorTop+(gheight*4);
     ng.ng_Width      = TextLength(&(screen->RastPort),
                                 HIDETEXT,
                                 strlen(HIDETEXT))
                      + OFFSET;
      ng.ng_Height     = gheight;
      ng.ng_GadgetText = HIDETEXT;
      ng.ng_GadgetID   = HIDE_GAD;
      ng.ng_Flags      = 0;
      g = CreateGadget(BUTTON_KIND, g, &ng, TAG_END);
      if (!g)  closedown(GAD_ERROR);
      ng.ng_LeftEdge   = 2*OFFSET + ng.ng_Width;
     ng.ng_Width      = TextLength(&(screen->RastPort),
                                 QUITTEXT,
                                 strlen(QUITTEXT))
                      + OFFSET;
      ng.ng_GadgetText = QUITTEXT;
      ng.ng_GadgetID   = QUIT_GAD;
      g = CreateGadget(BUTTON_KIND, g, &ng, TAG_END);
      if (!g)  closedown(GAD_ERROR);
      ng.ng_LeftEdge   = OFFSET;
      ng.ng_TopEdge    = screen->WBorTop+(2*gheight);
      ng.ng_Width      = TextLength(&(screen->RastPort),
                                 CYCLETEXT,
                                 strlen(CYCLETEXT))
                      + OFFSET;
      ng.ng_GadgetText = CYCLETEXT;
      ng.ng_GadgetID   = CYCLE_GAD;
      ng.ng_Flags      = PLACETEXT_ABOVE;
      g = CreateGadget(CYCLE_KIND, g, &ng,
          GTCY_Labels, DualLabels,
          GTCY_Active, DualIndex,
          TAG_END);
      if (!g)  closedown(GAD_ERROR);
      MMB_Window = OpenWindowTags(NULL,
          WA_Title,          MMB_NAME,
          WA_Gadgets,        MMB_Glist,
          WA_AutoAdjust,     TRUE,
          WA_InnerWidth,     ng.ng_Width+(2*OFFSET),
          WA_InnerHeight,    (4*gheight)+(2*OFFSET),
          WA_DragBar,        TRUE,
          WA_DepthGadget,    TRUE,
          WA_Activate,       TRUE,
          WA_SimpleRefresh,  TRUE,
          WA_IDCMP,          MMB_IDCMP,
          WA_PubScreen,      screen,
          TAG_END);
      if (!MMB_Window)  closedown(WIN_ERROR);
      GT_RefreshWindow(MMB_Window, NULL);
      FreeVisualInfo(vi);
      UnlockPubScreen(NULL, screen);
      }
   }
else {
   if (MMB_Window)  {
            struct IntuiMessage *imsg;
      while (imsg = GT_GetIMsg(MMB_Window->UserPort))
         GT_ReplyIMsg(imsg);
      CloseWindow (MMB_Window);
      FreeGadgets(MMB_Glist);
      MMB_Glist = NULL;
      MMB_Window = NULL;
      }
   }
}
/* ------------------------------- */
```

✓

Please write to:
Scott Palmateer
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Amazing Computing Gives You Great Reasons to Own an Amiga

Amazing Computing provides its readers with:
- In-depth reviews and tutorials
- Informative columns
- Latest announcements as soon as they are released
- Worldwide Amiga Trade Show coverage
- Programming Tips and tutorials
- Hardware Projects
- The latest non-commercial software

AC's TECH Gives You Great Reasons to Get Into Your Amiga

AC's TECH offers these great benefits:
- The only disk-based Amiga technical magazine
- Hardware projects
- Software tutorials
- Interesting and insightful techniques and programs
- Complete listings on disk
- Amiga beginner and developer topics

Order a SuperSub and get this great Amiga peripheral

—World's best authority on Amiga products and services
—Amiga Dealers swear by this volume as their bible for Amiga information
—Complete listings of every software product, hardware product, service, vendor, and even user groups
—Directory of Freely Redistributable Software from the Fred Fish Collection

12 Issues of Amazing + 2 AC's GUIDEs!

WINTER 1993 AC's GUIDE IS NOW AVAILABLE!

Using the audio.device

by Douglas Thain

Digitized and synthesized sounds can help to make games more enjoyable and utilities feel more professional. A simple beep emanating from a program just "feels" better than the Amiga's stark display flashing once. However, the audio examples provided in the ROM Kernel Manuals are less than clear cut. Even with these tools, audio support is not much use without the IFF standard sound format at your disposal. This article will help you to include synthesized and digitized sounds into your programs.

Audio Basics

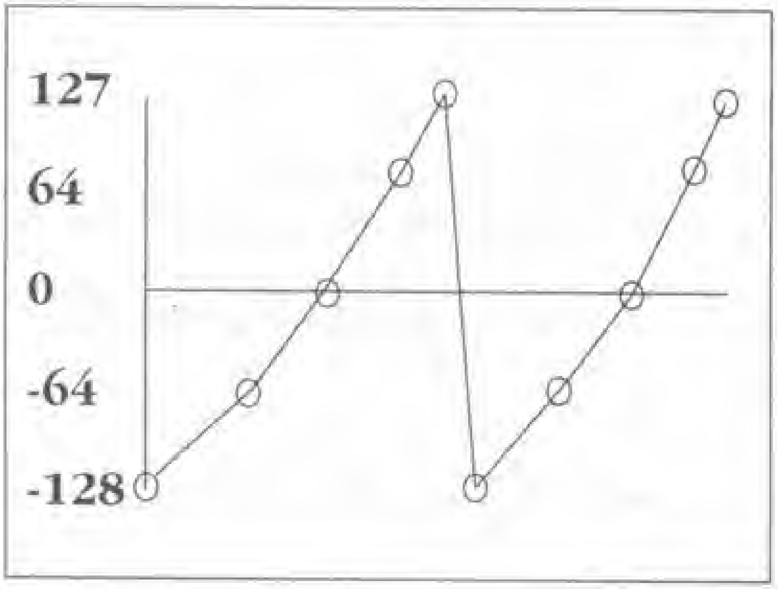
Electronic sounds can be divided into two main categories: digitized sounds and synthesized sounds. The Amiga's hardware treats the two in the same manner, but different programming approaches are needed for each type.

Both synthesized and digitized sounds are stored in chip memory—memory that the Amiga's custom chips have access to—as a series of integer values ranging from -128 to 127. Each discrete value is known as a sample. The entire piece of memory is also sometimes known as a sample, but we will refer to it as a waveform. When a waveform is played, the Amiga rapidly passes over the waveform and "clicks" the speaker in or out for each value. These clicks are turned into compressed air waves by the speaker, resulting in sound.

A synthesized sound is usually a small waveform that is repeated many times a second. Such sounds are often simple geometric patterns or mixtures of several such patterns. Figures 1and 2 are examples of synthesized waveforms. Synthesized sounds rarely sound like real sounds or instruments, but they take up very little memory and are easily created and manipulated in a program.

A digitized sound is a large sample—anywhere from a few kilobytes to several hundred—that is usually played once slowly. Such a sample is recorded using a digitizer. A digitizer measures the same sort of compression levels that a speaker emits, then feeds them to the computer. The recorded sample is then saved to disk for later use. Digitized sounds take some work to load and manipulate, but they allow for accurate reproduction of real sounds.

Figure One: Sawtooth Waveform



The Program

Listing 1, play.c is a program that demonstrates the playback of both digitized and synthesized sounds. It is run from the shell and takes one argument: the name of a sample or just "beep." If a name is provided, the program will attempt to load an IFF 8SVX file of that name and play it. "beep" will output a crude sawtooth waveform suitable for a terminal bell.

play.c can be compiled under any standard Amiga C compiler *except* DICE. DICE (as of version 2.05.15) does not support the function BeginIO(), which is essential to the program.

play.c should be compiled using long integers. Using Manx, this is:

```
cc +L play.c
ln play.o -lc32
```

play.c has three major components: playsample(), which sets up the device and plays a loaded waveform; loadsawwave(), which sets up a simple sawtooth waveform; and loadsample(), which loads sampled data from an IFF file. Let's take a close look at the important parts of these functions.

to Play IFF Sounds

Setting Up the audio.device

Before we play a waveform, we must do four things: secure a message port, allocate a request block, open the audio.device, and allocate an audio channel.

A new message port is necessary to communicate with the device. A nameless message port is created in the usual way:

```
struct MsgPort *port;
port = CreatePort(NULL,0);
if(port==NULL) puts("Couldn't open port!");
```

A request block is the vehicle that carries data between the device and our program. The request block we must use for the audio.device is an extended type known as struct IOAudio. This structure can be found in the include file devices/audio.h. This code will allocate an IOAudio:

```
struct IOAudio *request;
request = CreateExtIO(port,sizeof(struct IOAudio));
if(request==NULL) puts("Couldn't create request!");
```

Now we can open the device, specifiying our request block and the name of the device:

```
int error;
error = OpenDevice("audio.device",0,request,0);
if(error!=0) puts("Error opening audio.device!");
```

Before we can play a sound, we must allocate a channel. The channel allocation feature is provided so that several programs can compete for sound generation without blocking out important information for the user. To allocate a channel, a program must provide the channels desired along with a priority. These suggested sound priorities are taken from the ROM Kernel Manual: *Libraries and Devices*:

```
127      - Unstoppable
90 - 100 - Emergencies
80 - 90  - Annunciators (Bells and attention sig-
nals)
75       - Speech
50 - 70  - Sonic Cues
-50 - 50 - Music
-70 - 0  - Sound effects
-100 - -80      - Background
```

For example, a music program might allocate channels for its sounds at a priority of 25. An annunciator such as our program can allocate a channel at 85. This allocation would block out a particular channel just long enough to allow the annunciator to sound. The music program could then continue on with only a momentary loss of sound.

It is possible to use the device commands ADCMD_ALLOCATE and ADCMD_FREE to allocate and free many channels many times, but for a one-shot sound as we will require, it is often easier to use the automatic allocation and free features. If we set up a request block with allocation information *before* the device is opened, a channel is allocated right away, and our request block is ready to go. If we CloseDevice() without sending an ADCMD_FREE, the channel will be deallocated for us. Here's an example of automatic allocation:

```
#define LEFT0 1
#define RIGHT0 2
#define RIGHT1 4
#define LEFT1 8

int error;
UBYTE allocmap[] = { LEFT0|RIGHT0,
                     LEFT0,
                     RIGHT0 };

request->ioa_Request.io_Command = ADCMD_ALLOCATE;
request->ioa_Data              = allocmap;
request->ioa_Length            =
sizeof(allocmap);
request->ioa_Request.io_Flags    = ADIOF_NOWAIT;
request->ioa_Request.io_Message.mn_Node.ln_Pri =
85;

error = OpenDevice("audio.device",0,request,0);
if(error) puts("Couldn't open audio.device");
```

Our list of sets of channels (in order of preference) is in allocmap: first left and right channels 0, then each separately. If none of these are availible, OpenDevice() will fail. io_Command must be set to ADCMD_ALLOCATE, ioa_Data must point to the allocation array, and ioa_Length must be the size of the allocation array. The flag ADIOF_NOWAIT makes the action occur right away or fail. Otherwise, the command would wait until a channel was free. Finally, the priority of the message block must be set to our sound priority: 85 for annunciators.

Playing a Sample

Once the device is opened, playing a sample is relatively easy.

```
/* The clockspeed (in Hertz) of the custom chips */
#define COLORCLOCK 3579545

UBYTE  waveform[]  =  { 0, -64, -127, -127, -
64,0,64,128,128,128,64,0,-64,-127 };
int     wavesize = sizeof(waveform);
int     samplespersec = 500;
```

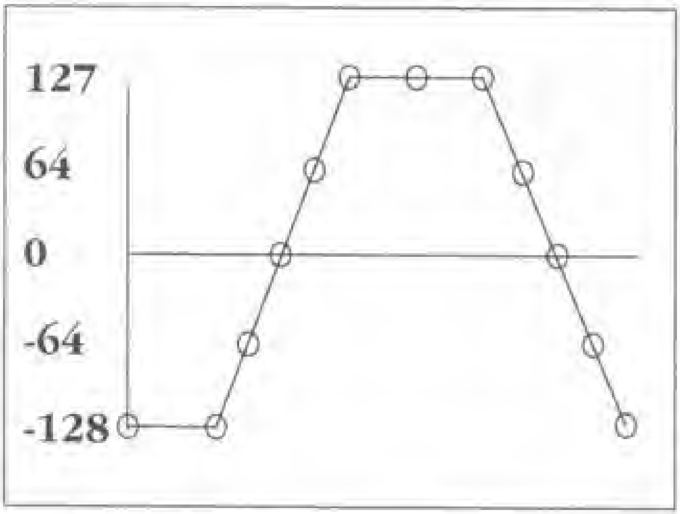
```
int      cycles  = 50;

request->ioa_Request.io_Command = CMD_WRITE;
requset->ioa_Request.io_Flags = ADIOF_PERVOL;
request->ioa_Data          = waveform;
request->ioa_Length = wavesize;
request->ioa_Period = COLORCLOCK/samplespersec;
request->ioa_Volume = 64;
request->ioa_Cycles = cycles;

BeginIO(request);
WaitIO(request);
```

CMD_WRITE says we are writing to the device. ADIOF_PERVOL tells the device to use the period and the volume settings in this request. Without ADIOF_PERVOL, the device would use the last values sent.

Figure 2: A "Squared-Triangle" Waveform



Waveform is a pointer to the data to be played, and wavesize is the length of the data. The waveform used is a mixture of a square wave and a triangle wave. (See Figure 2)

ioa_Period is the number of custom chip clock ticks between each sampled value. This value is rarely stored along with a waveform because it is so machine dependent. What is stored, however, is the number of sampled values per second that should be played. If we divide the number of custom chip ticks/sec (COLORCLOCK) by samples/sec, we get the number of ticks for each sample. (See Figure 3)

Volume ranges from 0 to 64 - the loudest possible suits us just fine.

ioa_Cycles is the number of times to loop through the waveform for a sampled sound, once plays the entire sound. A synthesized sound must play many cycles to be heard. Given the time in seconds to a play a synthesized sample, the number of cycles required is given by time*samplespersec/length. (See Figure 3)

The request is sent off to the device by using BeginIO(), and the waveform begins playing. Our program then waits for the sound to finish and the request to return by calling WaitIO(). Notice that SendIO() and DoIO() are generally not used with audio.device because they clear device specific flags such as ADIOF_PERVOL and ADIOF_NOWAIT. We must resort to the low level BeginIO() in order to preserve these flags.

Using Synthesized Sounds

The function loadsawwave() sets up a synthesized sound for playsample() to use. The waveform we will use is stored in sawwave. Notice that loadsawwave() allocates a new block of chip memory to copy sawwave into. We can't tell whether sawwave is in chip memory or not without allocating the memory ourselves. *It is essential that all waveforms be stored in chip memory.*

When creating your own synthesized sounds, there are two basic guidelines to follow:

- Make waveforms connect smoothly. The beginning and end of your waveforms should have similar values and slopes. Without this continuity, a waveform will click or pop each time it repeats.

- Make waveforms as much positive as negative. A waveform's volume is determined by the average absolute distance from the zero level. A waveform which is always on one side of the zero level will have little or no volume.

Loading IFF 8SVX Sounds

The Interchange File Format was developed by Electronic Arts and adopted by Commodore-Amiga as a general data standard. A complete discussion of the standard is far beyond the scope of this article and can be found in the ROM Kernel Manual: Libraries and Devices. However, a quick review of the 8SVX format is simple enough to develop a loader for our program.

Commodore has provided such tools as the iffparse.library to make manipulating IFF files easier. But for reading simple files as we will, it is easier to write our own routines and avoid making our program dependent on such libraries.

COLORCLOCK=3579545

$$\frac{\text{COLORCLOCK}}{\text{samplespersec}} = \frac{\text{ticks} \times \text{sec}}{\text{sample} \times \text{sec}} = \frac{\text{ticks}}{\text{sample}}$$

$$\frac{\text{Cycles} \times \text{length}}{\text{samplespersec}} = \frac{\text{cycles} \times \text{samples} \times \text{sec}}{\text{cycle} \times \text{samples}} = \text{Time (seconds)}$$

$$\frac{\text{Time} \times \text{samplespersec}}{\text{length}} = \text{Cycles}$$

The simplest IFF file is known as a FORM. A FORM can contain many different sub-parts, each known as chunks. Each chunk has a four letter identifier followed by a longword which shows the length of the chunk. A FORM is in fact itself a chunk, and can be made part of a larger IFF file. An example of this is an ILBM picture file as part of a larger ANIM animation file.

IFF identifiers are represented to humans as four letters, but the Amiga prefers to store the characters as longwords. For this purpose, we create the macro MakeID():

```
#define                    MakeID(a,b,c,d)
(((long)(a)<<24)|((b)<<16)|((c)<<8)|(d))

long num;

num = MakeID('F','O','R','M');
```

All this macro does is take four characters and packs them into one variable. When the program reads in a chunk ID, it reads in a longword and compares it to the longword created by MakeID().

Figure 4 shows the layout of a basic FORM 8SVX. An 8SVX can contain many chunks pertaining to the basic data. The author, creation notes, information for playing the waveform back as a musical instrument, and more can all be inside the file without getting in the way of the main data.

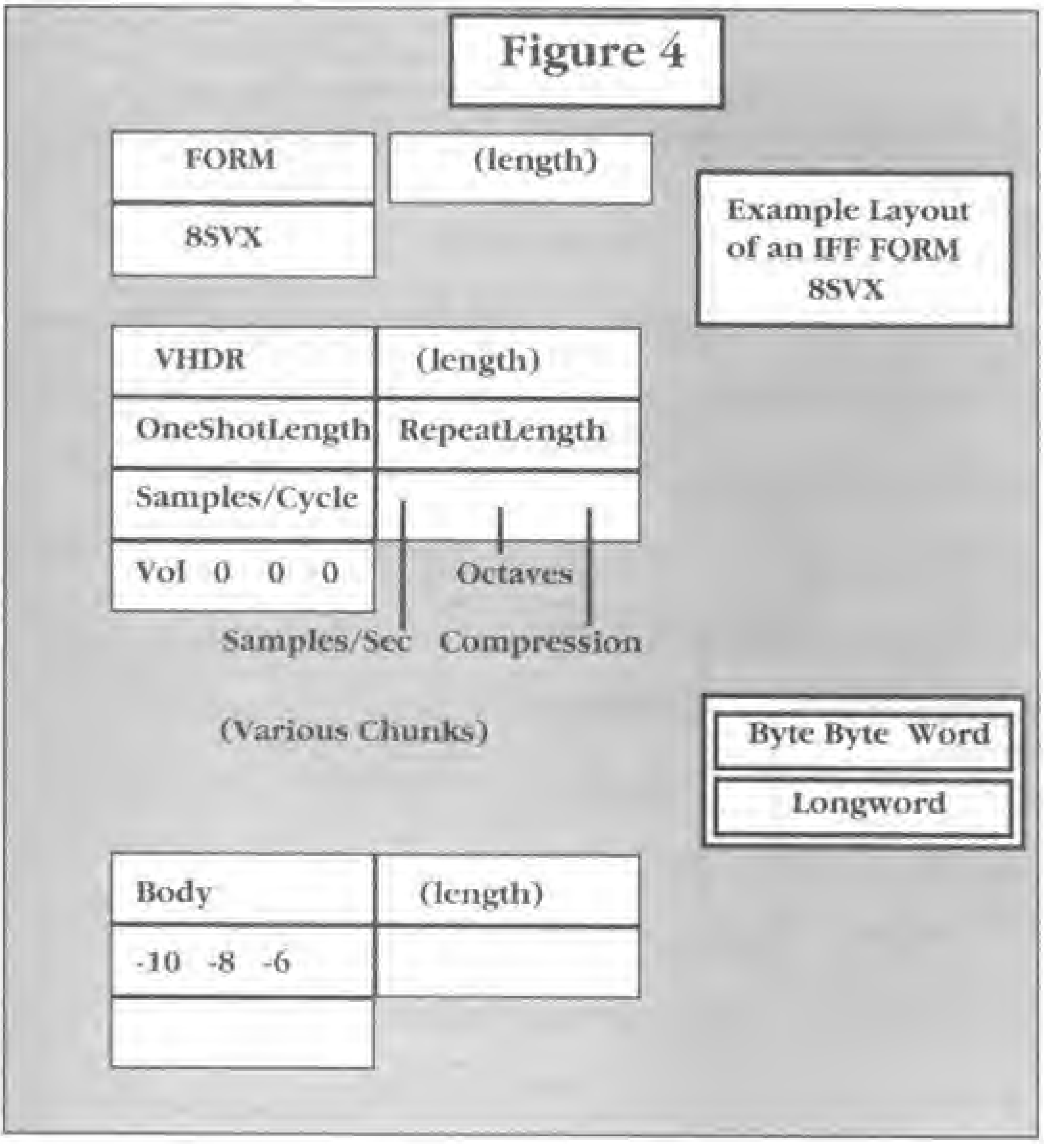
The two chunks we will be interested in are the VHDR and BODY chunks. The VHDR chunk only contains one field we need: the samples per second field (See Figure 4.) The BODY field contains only the raw sampled data. Our program must be flexible enough to skip over everything else that the file may contain.

Looking at Figure 4, here are the steps we must take to read in the waveform:

1. Make sure the first longword is FORM
2. Skip the next longword (the FORM length)
3. Make sure the next longword is 8SVX
4. Search for VHDR
5. Skip the next 4 longwords
6. Read in the next word as samples per second
7. Skip the next word
8. Search for BODY
9. Read in the waveform length
10. Read in the raw data

The function loadsample() performs all of this and is commented to show each step of the way.

And that's it! playsample() is perfect to be placed entirely inside another program—all the device work takes place completely inside of it. With the removal of global variables, loadsample() can be used as a general loader for all the sounds a program might need. As the program stands, it's perfect for playing sounds from an AmigaDOS script.



Listing 1

```
/* play.c
   by Douglas Thain

   Loads and plays an IFF 8SVX sample or does a
   simple beep.
   Compile using long ints - using Manx:

   cc +L play.c
   ln play.o -lcl32

  Standard Amiga code but can't compile under DICE - DICE
  doesn't have BeginIO() in c.lib or amigas.lib...
*/

#include <libraries/dos.h>
#include <devices/audio.h>
#include <exec/memory.h>

/* macros for IFF */
#define MakeID(a,b,c,d) ((long)(a<<24)|(b<<16)|(c<<8)|(d))
#define ID_FORM MakeID('F','O','R','M')
#define ID_8SVX MakeID('8','S','V','X')
#define ID_VHDR MakeID('V','H','D','R')
#define ID_BODY MakeID('B','O','D','Y')

/* audio channel definitions */
#define LEFT0  1
#define RIGHT0 2
#define RIGHT1 4
#define LEFT1  8

/* custom chip clockspeed */
#define COLORCLOCK 3579545

/* function prototypes */
BPTR           *Open();
void           *AllocMem();
void           *CreateExtIO();
void           *SetFunction();
struct MsgPort *CreatePort();
struct Library *OpenLibrary();

void           playsample(),loadsample();
void           loadsawwave(),closedown();

/* channel to use - in order of preference */
UBYTE allocmap[] = { RIGHT0,RIGHT1,LEFT0,LEFT1 };

/* sawtooth waveform to use if no sample specified */
UBYTE sawwave[] = { -127,-64,0,64,127 };

/* variables for sample data */
BPTR *file=NULL;
UBYTE *waveform=NULL;
LONG  wavesize;
UWORD samplespersec;
UBYTE cycles;

main(argc,argv)
int argc;
char *argv[];
{
   /* If no argument, show command format */
   if(argc!=2) {
    puts("Usage: play <samplename|beep>");
      exit(20);
       }

   /* If argument was "beep", setup sawwave.
      Otherwise, load sample */
  if(!strcmp(argv[1],"beep")) loadsawwave();
   else                       loadsample(argv[1]);

   /* Do it! */
  playsample();

   /* exit */
  closedown(0);
}

/* closedown() cleans up only structures that were used */
void closedown(err)
int err;
{
  if(waveform!=NULL) FreeMem(waveform,wavesize);
  if(file!=NULL)     Close(file);
  if(err==15)        puts("Not a valid IFF 8SVX file!");
  exit(err);
}

void loadsawwave()
{
   /* sawwave must be copied into chip memory */
  waveform=AllocMem(sizeof(sawwave),MEMF_CHIP|MEMF_PUBLIC);
  if(waveform==NULL) closedown(20);

  strncpy(waveform,sawwave,sizeof(sawwave));
   samplespersec = 300;
   cycles = 40;
  wavesize = sizeof(sawwave);
}

/* loadsample() can massage and load a FORM 8SVX but does
  not understand conglomerate IFF files */
void loadsample(name)
char *name;
{
   int  size,x;
   LONG tmp;

   /* nab the file passed to us */
  file = Open(name,MODE_OLDFILE);
  if(file==NULL) {
    puts("Couldn't open file!");
     closedown(20);
      }

   /* First longword must be "FORM" */
  size = Read(file,&tmp,sizeof(LONG));
  if(size!=sizeof(LONG)) closedown(15);
  if(tmp!=ID_FORM) closedown(15);

   /* next is discarded */
  size = Read(file,&tmp,sizeof(LONG));
  if(size!=sizeof(LONG)) closedown(15);
```

```c
    /* next must be "8SVX" */
    size = Read(file,&tmp,sizeof(LONG));
    if(size!=sizeof(LONG)) closedown(15);
    if(tmp!=ID_8SVX) closedown(15);

    /* now find a "VHDR" */
    while(1) {
      size = Read(file,&tmp,sizeof(LONG));
      if(size!=sizeof(LONG)) closedown(15);
      if(tmp==ID_VHDR) break;
    }

    /* first 4 longwords are no use to us */
    for(x=0;x<=3;x++) {
      size = Read(file,&tmp,sizeof(LONG));
      if(size!=sizeof(LONG)) closedown(15);
    }

    /* the next word is sampling rate */
    size = Read(file,&samplespersec,sizeof(UWORD));
    if(size!=sizeof(UWORD)) closedown(15);

    /* next word is no use to us */
    size = Read(file,&tmp,sizeof(UWORD));
    if(size!=sizeof(UWORD)) closedown(15);

    /* now find a "BODY" */
    while(1) {
      size = Read(file,&tmp,sizeof(LONG));
      if(size!=sizeof(LONG)) closedown(15);
      if(tmp==ID_BODY) break;
    }

    /* next longword is size of sample */
    size = Read(file,&wavesize,sizeof(LONG));
    if(size!=sizeof(LONG)) closedown(15);

    /* can we allocate that much memory? */
    waveform =
      AllocMem(wavesize,MEMF_CHIP|MEMF_CLEAR|MEMF_PUBLIC);
    if(waveform==NULL) closedown(16);

    /* read in as much as we can get */
    size = Read(file,waveform,wavesize);
    /* as long as we got SOMETHING, we'll play it */
    if(size==0) closedown(15);

    /* and close the file */
    Close(file);
    file = NULL;

    cycles = 1;
    return;
}

void playsample()
{
    struct MsgPort *port;
    struct IOAudio *req;
    int error=1;

    /* create a no-name port */
    port = CreatePort(NULL,0);
    if(port!=NULL) {

      /* allocate a request block */
      req = CreateExtIO(port,sizeof(struct IOAudio));
      if(req!=NULL) {

        /* set up the block so that OpenDevice() will
           automatically allocate a channel from
           allocmap() for us */
        req->ioa_Request.io_Message.mn_Node.ln_Pri = 85;
        req->ioa_Request.io_Flags = ADIOF_NOWAIT;
        req->ioa_Request.io_Command = ADCMD_ALLOCATE;
        req->ioa_Data = (UBYTE *) allocmap;
        req->ioa_Length = sizeof(allocmap);

        error = OpenDevice("audio.device",0,req,0);
        if(error==0) {
          /* set up the block with our data */
          req->ioa_Request.io_Command = CMD_WRITE;
          req->ioa_Request.io_Flags = ADIOF_PERVOL;
          req->ioa_Data   = waveform;
          req->ioa_Length = wavesize;
          req->ioa_Period = COLORCLOCK/samplespersec;
          req->ioa_Volume = 64;
          req->ioa_Cycles = cycles;

          /* begin the sound  (DoIO() or SendIO() can't
             be used - they will corrupt io.Flags) */
          BeginIO(req);

          /* wait for it to finish */
          WaitIO(req);

          /* close everything up */
          CloseDevice(req);
        }
        DeleteExtIO(req);
      }
      DeletePort(port);
    }
    if(error) puts("Error opening audio.device");
    return;
}
```

Please write to:
Douglas Thain
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Make Your Own 3-D Vegetation Objects

by Laura M. Morrison

An earlier article described how images can be built up of 'tiles' using iterated function systems to create self-similar graphics. See Reference 1. It gave a way of placing tiles for 2-D images using mouse clicks, and showed how to use Cramer's rule to get the transformation coefficients. Although the same method should work for placement of 3-D tiles to get 3-D transformations, mouse clicking on a flat screen for 3-D placement can be imprecise and confusing. This article describes a different way of getting transformations for 3-D self-similar iterated function system objects.

The transformations that place tiles and whose iteration produce self-similar objects, are known to be composites of translation, scaling and rotation. With a knowledge of these simple transformations, the computer artist, working from his idea, or sketch, of how he wants the completed object to turn out, can easily write specifications ('specs') describing each tile's placement.

Illustration 1. pictures the relationships used to derive equations for these transformations. For simplicity they are given for the x-y plane only. The equations have the general form:

```
P'(x',y',z') =
        f(P(x,y,z),Sx,Sy,Sz,h,j,k,(xf,yf,zf))
```

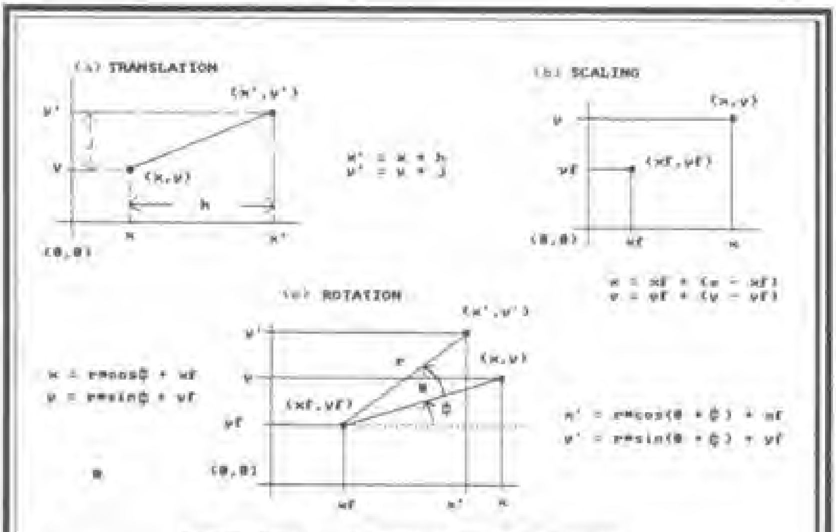


ILLUSTRATION 1. TRANSFORMATION RELATIONSHIPS

where P' is the transformed point, Sx,Sy,and Sz are the angles of rotation about the x-,y-,z-axis (or lines parallel with these axis); h,j,and k are translations along the x-,y- and z-axis; and Pf(xf,yf,zf) is the fixed point. The transformations used in functions systems are composites of these three simple planar transformations.

Illustration 1(a) shows the translation relationships. The coordinates of the transformed point, P'(x',y',z'), are the coordinates of the original point, P(x,y,z), but shifted up or down, right or left, backwards or forwards, by adding quantities h to x, j to y, and k to z. The equations are:

```
x' = x + h
y' = y + j
z' = z + k
```

Most discussions of scaling and rotation assume the origin as the fixed point. That form is not useful when applying the transformations, as in tiling, to parts of a complex whole. It is desirable to be able to specify the fixed point of the transformation. More useful equations that include a non-zero fixed point can be derived as follows:

Illustration 1(b) depicts the scaling relationships. Scaling changes the distances between points. One point always remains fixed and its distance to each other point changes by a scale factor. The x, y, and z distances change independently and the scale factors for x, y, and z need not be the same. To derive expressions for the coordinates of the scaled point P'(x',y',z'), given a specific fixed point, Pf(xf,yf,zf), we can start with identity:

```
x = xf + (x - xf)
y = yf + (y-yf)
z = zf + (z-zf)
```

These simple identities expresses any point in space in terms of its distance from the fixed point. The transformed point will want to have the same form but the distance from P to Pf scaled up or down. Multiplying the distance between points by a scale factor gives:

```
x' = xf + Sx*(x-xf)
y' = yf + Sy*(y-yf)
z' = zf + Sz*(z-zf)
```

(This checks when (xf,yf,zf) = (0,0,0), or when (xf,yf,zf) = (x,y,z).) Clearing and rearranging terms results in the forms:

```
x' = Sx*x + xf*(1-Sx)
y' = Sy*y + yf(1-Sy)
z' = Sz*z + zf(1-Sz)
```

where the second term of each equation is constant, the same for all points P(x,y,z).

Illustration I(c) depicts rotation relationships. We can use the formula for the sine and cosine of the sum of two angles:

```
cos(# + phi) = cos#*cos(phi) - sin#*sin(phi)
sin(# + phi) = sin#*cos(phi) + cos#*sin(phi)
```

and the identities:

```
x = r*cos(phi) + xf
y = r*sin(phi) + yf.
```

which, when rearranged, give:

```
r*cos(phi) = x - xf
 r*sin(phi) = y - yf.
```

Referring to the diagram we find an equation for the transformed point is:

```
x' = r*cos(# + phi) + xf
y' = r*sin(# + phi) + yf
```

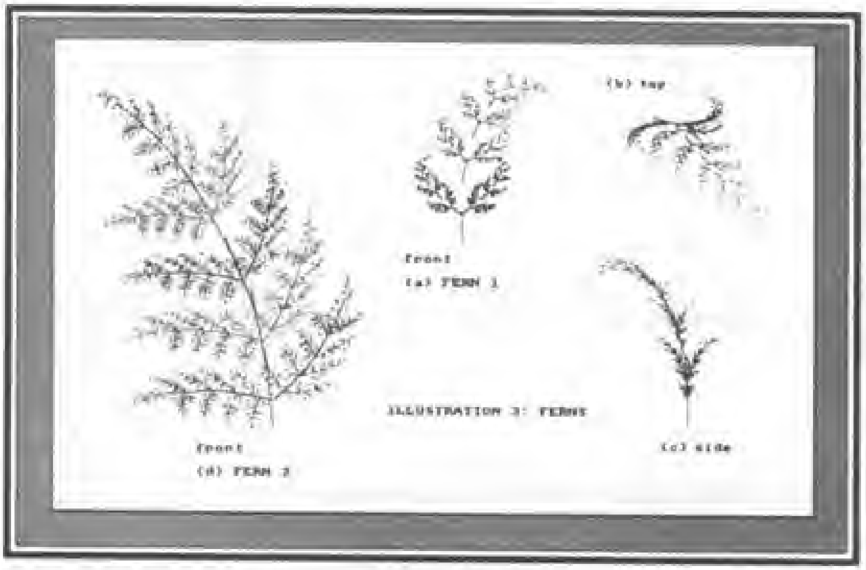
Substituting, then rearranging gives the final expressions for the coordinates of the point in the x-y plane rotated about a line parallel with the z axis through the fixed point:

```
x' = xf + (x-xf)*cos# - (y-yf)*sin#
y' = yf + (y-yf)*cos# + (x-xf)*sin#
```

Analogous formula hold for transformed points in the x-z plane and the z-y plane.

Trigonometry and Analytic Geometry are the highest mathematics needed. See any introductory Analytic Geometry text (e.g., Reference 2).

Each tile requires 12 numbers (zeros count) to specify the composite transformation that maps the 3-D whole into the 3-D tile. These specifications, or 'specs', describe directly what the transformation must do. The user writes four sets of three numbers to an ASCII file to be read by the 3-D_IFS_decoder program. Table 1 lists the specifications for placement of tiles of Fern 1, shown in Illustration 3(a), (b), and (c). The first six numbers are the x, y, and z scales and offsets. The determine size and position of plots on the screen and are not part of the transformations. Each tile requires four sets of three numbers.



The fourth set of three numbers give the coordinates of the scaling fixed point. They are the same for each tile of Fern 1. They are:

```
160.0    350.0    160.0
```

The numbers refer to Amiga screen coordinates. Although these are integer pixel amounts they are written in floating point because the program calculates in floating point. (The fixed point of the rotations is calculated by 3-D_IFS_decoder program.)

The second set of three numbers are the three scaling factors. They describe what percent of the whole the size of the tile's x, y, and z dimensions should be. For example, a tile to be 3/10 the width of the whole, 4/10 its height, and 3/10 its depth would have the set of scale factors:

```
0.3      0.4      0.3
```

Note that the size of the whole is only implicitly defined by the system of transformations. The scale factors give relative sizes.

The first set of three numbers for each tile in the 'specs' file are the amounts to move the the tile to bring it into position for rotation into its final position. For example to translate 0 pixels in the x direction, -15 pixels in the y direction, and 0 pixels in the z direction the 'specs' would be:

```
0.0     -15.0    0.0
```

Since the fixed point has y = 350 near the bottom of the screen. Adding -15 to y transforms the y value to y = 335, up the screen. The scaled tile is to be moved 15 pixels up.

The third set of three numbers are three angles. They tell how much to rotate about the x-axis, y-axis, and z-axis (or, lines parallel with the x-, y-, or z-axis) respectively, to place the tile in its final position in the whole. For example, to rotate the tile 180 degrees about line parallel with the y-axis, 0 degrees about a line parallel to the x-axis and 0.9 radian clockwise about the z-axis the 'specs' are:

```
0.0      3.142    -0.9
```

Positive numbers rotate counterclockwise, negative numbers rotate clockwise. The angles of rotation are expressed in radians. They will have values between 0.0 and 2*3.1418..

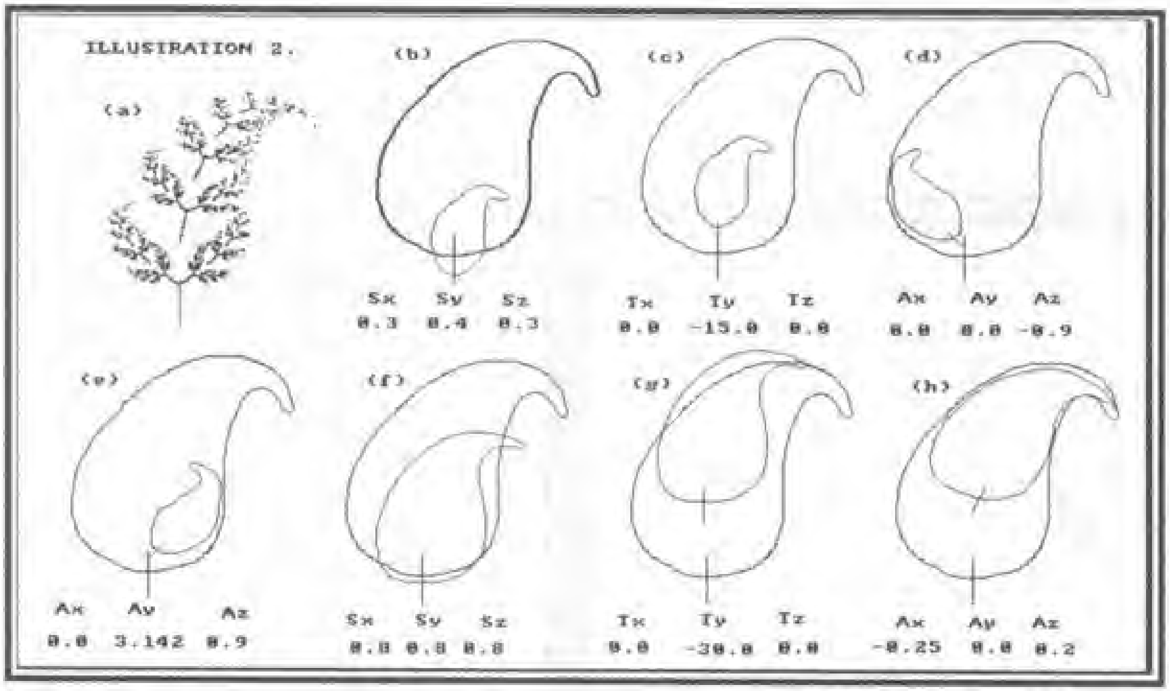
In writing specifications for a tile it may be helpful to write a general description first. Give a verbal account of how the computer is to transform the whole in order to get the tile. For example, "... Make a tile a little less than half the size of the whole and proportionately thinner. Make it branch out from the main stem only a little way up the stalk. Also let it tilt backwards slightly." Then, translate this description into the twelve numbers. Learning the basic information in Illustration 1 (a), (b), and (c) can also be helpful to writing specifications.

During iteration the decoder program performs the component transformations in sequence instead of by evaluating one vast complex formula. Such a formula could be derived by substituting, combining terms and clearing expressions. It would be a single transformation with coefficients in three-dimensions analogous to the coefficients defined by mouse clicks for the two-dimensional iterated function systems.

The order in which component transformations are applied makes a difference. The transformations are not commutative. Different orders give different results. The order chosen here is easy to learn and it simplifies, clarifies, and routinizes 'specs' selection. It provides a routine procedure for the user to follow while learning to imagine the desired image and express its tiles in terms of transformations of the whole. This order is appropriate to defining tiles for vegetation such as ferns and trees as described here. At some point the user may want to experiment with the order of transformations if only to check what difference the order makes.

Illustration 2 depicts the sequence of scale, translate, and then rotate as the results of the corresponding specifications during the placement of tiles of Fern 1. The scaling fixed point is the lower tip of Fern 1 (a). First scale the whole with fixed point the bottom most point of the whole, as in (b). Next translate the tile to position for rotation as in (c). Last, rotate the tile about a line parallel with the z-axis as in (d). The next tile is defined similarly except it was rotated 180 degrees about the y-axis to flip it over before slanting it clockwise 0.9 radians as in (e). The large top tile was scaled, (f), translated, (g), and rotated about a line parallel with the z- axis 0.2 radians and about a line parallel with the x-axis -0.25 radians, (h).

Illustration 2(a) shows a different fern, Fern 2, built up completely of miniature models of the whole. Table 2, gives the specifications for Fern 2 as they would be submitted to 3-D_IFS_decoder program. Note that when a segment of the stem bends the next segment must be translated to join it. This is different from Fern 1 which seemingly adjusts automatically. Fern 1 exploits the sub-tiles of the large top tile. Besides explicit tiles the transformations produces sub-tiles whose placement can be exploited to build up an object. Unlike Fern 2, which is composed of many tiles, Fern 1 was built of only three. One tile was specified as a much larger replica of the whole so the transformations produced a fern of seemingly many tiles. Note that each segment of the bent stem has been placed correctly and automatically by the transformations. Since Fern 2 was built up of many same-size tiles each segment of stem had to be specifically translated to sit on the segment below. As described so far, Fern 2 is two-dimensional. Its orthogonal projections on the x-z and z-y planes are lines. The user can experiment by giving the fern a twist about the vertical or bending it backward or forward to see the effect on the projections.



The 3-D_IFS_decoder program (Listing 1) uses the specifications to produce the points of a 3-D object. To run the program first prepare the 'specs' file as described above for the fern. It should contain the screen scale and offset values and then four sets of three numbers for each tile. (Don't forget to add an end-of-file 999.9's set.) Place the 'specs' file in the same directory as the program. At the CLI prompt, type:

3-D_IFS_decoder <output_image_filename> The program reads the 'specs' file and randomly iterates the transformations described by the specs. It draws three screen images, orthographic projections, of the object's front on the x-y plane, of the top on the x-z plane and of the side on the y-z plane.

3-D_IFS_decoder writes the screen images to RAM files, in raw format. They must be converted to ILBM format using Commodore's public domain program 'raw2ilbm'. To convert to ILBM format put a copy of 'raw2ilbm' in the RAM: directory, CD to RAM: and type at the CLI prompt: raw2ilbm <filename> df1:<filename.pic> hi 4 For example. This will add 'pic' suffix and put the ILBM file on the disk in the external drive, df1:.

To avoid large files that cannot be opened or even contained on one disk, the program writes the object point set to a series of files:

```
<output_image_filename>set.1;
<output_image_filename>set.2
...; etc.
```

It writes only so many points of the object to a numbered ('outfileN') file, closes it and opens a new numbered file ('outfileN+1'). The set of points that make up the object are written with the format of four 16-bit, short integers:

```
color register x, y, z ;
```

The user can make these files any convenient size. The number of points the object will have is determined by how long the user allows iteration to continue. Since the points are randomly scattered their number merely determines the density of the object's point set.

The program continues until the user stops it by clicking in the CloseWindow gadget on the small "idcmp" window in the x-y screen. Clicking the CloseWindow gadget saves the set of object points and the three images to RAM: Or, the user may key 'q' to quit saving only the object points.

3-D_IFS_decoder writes, as does each program of this article, a record of its activity to a file in the same directory called 'Run_Notes.' 3-D_IFS_decoder saves relevant run data here as well as a user supplied

Table 1

Specifications for Fern 1.

1.5	0	1.3	-200	1.1	0
0.0	15.0	0.0	0.3	0.4	0.3
0.0	0.0	[illegible]	200.0	330.0	160.0
0.0	[illegible]	0.0	0.3	0.4	0.3
0.0	[illegible]	0.0	200.0	150.0	160.0
0.0	0.0	0.0	0.8	0.8	0.8
0.0	0.0	[illegible]	400.0	350.0	[illegible]
0.0	[illegible]	0.0	0.8	0.6	0.8
[illegible]	[illegible]	[illegible]	200.0	150.0	160.0
0.0	[illegible]	0.0	0.8	0.6	0.8
-0.25	[illegible]	0.2	200.0	150.0	160.0
999.9	999.9	999.9	999.9	999.9	999.9
999.9	999.9	999.9	999.9	999.9	999.9

comment at the end of the run. The user might want to add times and dates to Run_Notes.

Run_Notes collects information about activity in the directory. When it becomes too long the user may want to archive it. Its great to know information is available if needed. No more "how was that produced?"

TABLE 2

Specifications for Fern 2

0.0	0	1.0	0	1.0	0
10.0	-125.0	.0	0.85	0.4	0.35
0.0	0.0	-7.0	160.0	350.0	180.0
0.0	-125.0	0.0	0.4	0.4	0.35
0.0	0.0	0.0	160.0	350.0	160.0
0.0	0.0	0.0	1.0	0.24	7.0
0.0	0.0	-0.1	160.0	350.0	160.0
0.0	[illegible]	0.0	0.22	0.4	0.22
0.0	0.0	[illegible]	160.0	350.0	180.0
-7.1	-70.0	0.0	0.09	0.4	0.22
0.0	0.0	0.0	160.0	180.0	180.0
[illegible]	55.0	0.0	0.0	0.24	5.0
0.0	0.0	[illegible]	160.0	350.0	160.0
-25.0	-110.0	0.0	0.2	0.38	3.0
0.0	0.0	[illegible]	160.0	350.0	180.0
-10.0	-125.0	2.0	0.0	0.30	0.0
0.0	0.0	0.0	160.0	350.0	160.0
-15.0	-108.0	0.0	0.0 0.	.24	0.0
0.0	0.0	-0.5	160.0	350.0	160.0
-10.0	-115.0	0.0	0.2	0.3	0.2
0.0	0.0	-1.0	160.0	350.0	160.0
-10.0	-125.0	0.0	0.2	0.1	0.2
0.0	0.0	0.0	160.0	350.0	160.0
-15.0	-100.0	0.0	0.0	0.24	0.0
0.0	0.0	-0.5	160.0	350.0	180.0
-65.0	-100.0	0.0	0.125	0.4	0.28
0.0	0.0	[illegible]	160.0	350.0	160.0
999.9	999.9	999.9	999.9	999.9	999.9
999.9	999.9	999.9	999.9	999.9	999.9

TABLE 3

Specifications for Tree.

0.0	100	0.0	0	0.8	1.0
0.0	-115.0	0.0	0.6	0.38	0.0
0.0	1.0	-0.1	160.0	350.0	160.0
0.0	-115.0	0.0	0.6	0.5	0.6
-7.3	1.87	-0.4	160.0	350.0	160.0
0.0	-187.0	0.0	0.8	0.5	0.8
0.2	-8.0	0.6	160.0	350.0	160.0
0.0	-88.0	0.0	0.59	0.5	0.59
-0.7	0.2	+0.6	160.0	350.0	160.0
0.0	-10.0	0.0	0.25	0.4	0.25
0.7	0.3	0.8	160.0	350.0	160.0
0.0	-30.0	0.0	0.6	0.8	0.6
0.0	-1.87	0.0	160.0	350.0	160.0
0.0	5.0	0.0	0.02	0.5	0.02
0.0	0.0	0.0	160.0	350.0	160.0
0.0	0.0	0.0	0.02	0.3	0.02
0.0	0.0	0.0	160.0	350.0	160.0
999.9	999.9	999.9	999.9	999.9	999.9
999.9	999.9	999.9	999.9	999.9	999.9

Since 'List' has come to have a specific structured meaning it may be better not to call the object point set a list but rather a "point-set". Each point is 'free standing' and does not follow any other point. Sets of points for different objects can be combined to make a one object by simply adding sets of points together.

Illustration 4(a), (b), and (c) shows a 3-D point-set tree: top, front, and side, made with the 3-D_IFS_decoder program. The 'specs' file for the tree is given in Table 3.

RotaView (Listing 2) reads a file of object points produced by 3-D_IFS_decoder and displays an orthographic projection on the x-y plane. The user can perform sequences of transformations by keying instructions. By keying 'a', 'b', or 'g' the user instructs RotaView to rotate the object about a line parallel to the x-, y- or z-axis and redraw

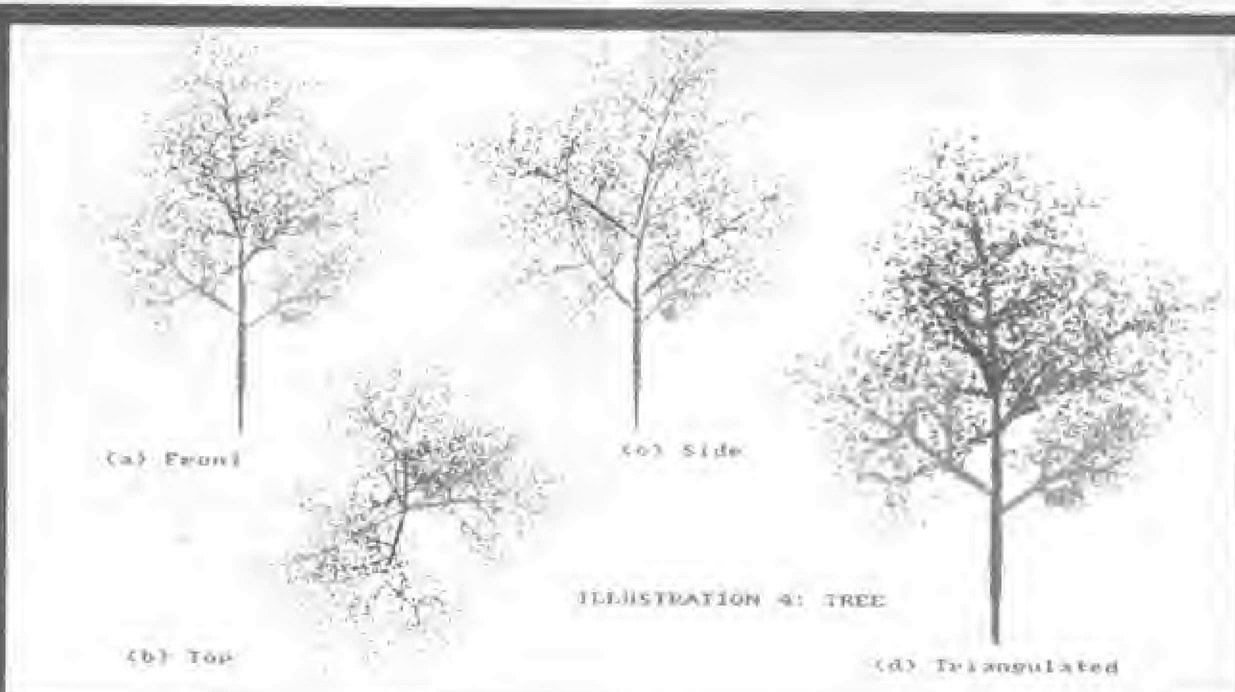


ILLUSTRATION 4: TREE

its x-y projection. Keying '-' before the angle letter reverses the direction of rotation. If the user keys 's' the program saves the transformed image and the rotated object point set.

RotaView also reads a parameter file 'rvparams.' The rvparams file contains the coordinates for the fixed point, the angle increments and scale and offset values. The scale and offsets only apply to the screen coordinates, not the points of the rotated image. Table 4(a) lists the 'rvparams' file for the tree.

TABLE 4

Parameter files

[illegible]

Rotaview can be used to view any 3-D point-set object, not just vegetation. Try, for example, a wire cube point-set included on disk. It could also be used to easily produce an animation sequence for a rotating object.

For the 3-D vegetation objects to be useful they must be converted to input for a 3-D program. The next two programs, SculptData and SDtoSC3D do that.

Sculpt3D was chosen to import the 3-D vegetation objects because it is inexpensive, its file format readily available, and because there are, reportedly, public domain programs to convert Sculpt3D's files to any other 3-D program's format. See Reference 3. Sculpt3D works with triangular faces so the object's points must be transformed into tiny triangular faces. SculptData program does this.

SculptData (Listing 3) reads an object point-set produced by 3-D_IFS_decoder and expands each point into three points. It selects three nearby points at random. It writes the three points and indices to three files as raw data for vertices, edges, and faces. SculptData writes the final count of vertices, edges, and faces to Run_Notes.

SculptData reads a user prepared parameter file called 'sdparams.' The first parameter is the edgelength which determines the distance between the triangle of points. The second parameter is the number of object points to expand. This number determines the object's density and should be made as large as available memory permits. Table 4(b) lists the 'sdparams' file for the tree.

Illustration 4(d) shows the front projection of the same tree as in (a), (b), and (c), where each point has become three points. (This expanded point-set was collected from SculptData as a by-product as it worked on its three files. To collect this data add another output file and three 'fprintf' statements to the code.) The 'edgelength' for the tree in Illustration 4(d) was two. The edgelength for the tree to be imported to Sculpt3D was four.

TABLE 5

Run_Notes - Record of program activity during production of tree '.scene' input file for Sculpt3D.

[illegible]

SDtoSC3D (Listing 4) reads the three raw data files produced by SculptData and converts them to one file in Sculpt3D's '.scene' format. SDtoSC3D also reads a user prepared parameter file 'tiparams.' Table 4(c) lists the parameter file used to prepare the '.scene' file for the tree. The first three numbers are the numbers of vertices, edges and faces on the input files. (Available on Run_Notes.) The next number multiplies point coordinates. A factor of 100 was found to be necessary

to get Sculpt3D to accept the 'scene' file. In the parameter file the user also specifies the colors to be used corresponding to the register numbers of the raw data.

Illustration 5 shows the tree as rendered by Sculpt3D (or, as converted by Transfer from Sculpt3D's HAM image to lo-res. The HAM image is on disk.) The STACK was set at 200000 and Sculpt3D was loaded from CLI. The image took about 26 hours to render on an Amiga A1000 with 2.5 M RAM.

Table 5 lists the Run_Notes record of the tree's production.

Listing 1

```
/* LISTING 1:  3-D_IFS_decoder
Copyright 1992 by Laura M. Morrison */

#include "stdio.h"
#include "libraries/dosextens.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "exec/exec.h"
#include "math.h"
#define COLORS12    32L
#define MAXTRANS    16L
#define DEPTH        4L
#define WIDTH      640L
#define HEIGHT     400L
/* These colors help tell which transformation
produced which tiles. */
USHORT cmap[] = { 0x000,0xf00,0x0f0,0x00f,
                              0xff0,0x0ff,0xf0f,0xfff,
                              0x800,0x080,0x008,0x880,
                              0x088,0x808,0x888,0x444 };
/* Set up for three screens for the three
orthogonal projections of the object on the x-y
plane, the x-z plane, and the z-y plane. */
struct NewScreen newscrxy = {
   0,0,640,400,4,0,1,
   HIRES|LACE,
   CUSTOMSCREEN|CUSTOMBITMAP,
   NULL,"x-y plane ",NULL,NULL };
struct Screen *scrxy;
struct NewScreen newscrzy = {
   0,0,640,400,4,0,1,
   HIRES|LACE,
   CUSTOMSCREEN|CUSTOMBITMAP,
   NULL," z-y plane",NULL,NULL };
struct Screen *scrzy;
struct NewScreen newscrxz = {
   0,0,640,400,4,0,1,
   HIRES|LACE,
   CUSTOMSCREEN|CUSTOMBITMAP,
   NULL," x-z plane ",NULL,NULL };
struct Screen *scrxz;
struct NewWindow nw = {
   500,420,100,10,0,1,
   CLOSEWINDOW|VANILLAKEY,
   WINDOWCLOSE|ACTIVATE|WINDOWDEPTH|WINDOWDRAG,
   NULL, NULL, "Ldump", NULL, NULL,
   0,0,100,10, CUSTOMSCREEN };
struct Window *w;
UBYTE *buff;
struct FileHandle *Open();
struct BitMap *bitmxy;
struct BitMap *bitmzy;
struct BitMap *bitmxz;
struct RastPort *rpxy;
struct RastPort *rpxz;
struct RastPort *rpzy;
struct ViewPort *vpxy;
struct ViewPort *vpxz;
struct ViewPort *vpzy;
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
static float  ax[MAXTRANS],  ay[MAXTRANS];
static float  az[MAXTRANS],  bx[MAXTRANS];
static float  by[MAXTRANS],  bz[MAXTRANS];
static float  cx[MAXTRANS],  cy[MAXTRANS];
static float  cz[MAXTRANS],  xt[MAXTRANS];
static float  yt[MAXTRANS],  zt[MAXTRANS];
static float  xx, yy, zz, x, y, z;
static float xs, ys, zs, xl, yl, zl;
static float xxa, yya, zza, xxr, yyr, zzr;
static float cosa, sina, cosb;
static float sinb, cosg, sing;
static float cxt, cyt, czt;
static float xl, yl, zl;
static float xoffset, yoffset, zoffset;
static float xscale, yscale, zscale;
static char outfile[120];
static char str[80];
static char comment[120];

main(argc, argv)
int argc;
char *argv[];
{
         LONG xtemp, ytemp, ztemp;
         struct IntuiMessage *msg;
         ULONG class;
         USHORT code;
         LONG count, entry, next;
         int nt, number, count, count2;
         int seed, r, i, ntrans;
         FILE *fopen(), *fp, *fo, *fp, *fp;
         LONG planesize, planes;
         colorsize = COLORS12;
      if (argc < 2 )
        {
        printf("Need output image filename.\n");
         exit(10);
        }
         IntuitionBase = (struct IntuitionBase *)
          OpenLibrary("intuition.library",0);
      if((IntuitionBase == NULL)
        {
         printf("Couldn't open Intuition library\n");
         exit(11);
        }
         GfxBase = (struct GfxBase *)
                  OpenLibrary("graphics.library",0);
         if(GfxBase == NULL)
         {
         printf("Couldn't open graphics library\n");
         CloseLibrary(IntuitionBase);
         exit(12);
          }
         printf("\n\n\n\n\n\n");
         printf("                    ");
         printf("3-D IFS DECODER  Program \n\n");
         printf("                    ");
         printf("Copyright 1992 by Laura M. Morrison");
         printf("\n\n\n\n\n\n\n\n");
         /* Read in scales and offsets as necessary to
         set the picture center on the Amiga screen. */
         if ((fp = fopen("param","r")) == NULL)
         {
         printf("Could not open parameter file.\n");
         exit(13);
         }
      fscanf(fp," %f  %f ",&xscale,&xoffset);
      fscanf(fp," %f  %f ",&yscale,&yoffset);
         fscanf(fp," %f %f ",&zscale,&zoffset);
         i=0;
         /* Read in the (ascii) file of the placements.*/
ntrans=i;
         fscanf(fp," %f %f ", &ax[i], &ay[i]);
         fscanf(fp," %f %f ", &az[i], &bx[i]);
```

```
fscanf(pp," %f %f ", &ys[i], &zs[i]);
fscanf(pp," %f %f ", &xa[i], &ya[i]);
fscanf(pp," %f %f ", &za[i], &xf[i]);
fscanf(pp," %f %f ",  &yf[i], &zf[i]);
if( xt[i] < 998.0 )
{
          i++;
          if( i > (MAXTRANS-1) )
          {
printf("Need more storage.\n");
                    goto quit;
          }
goto readmore;
}
ntrans = i;
fclose(pp);
/* Set up the super bitmaps and open the
three screens. */
planesize = (LONG)(WIDTH/8)*HEIGHT;
bitmxy = (struct BitMap *)AllocMem
          (sizeof(struct BitMap),MEMF_CHIP);
if (bitmxy==NULL)
{
printf("No memory for xy bmap struct.\n");
goto quit;  }
InitBitMap(bitmxy,DEPTH,WIDTH,HEIGHT);
for (i = 0; i < DEPTH; i++)
{
bitmxy->Planes[i]=(PLANEPTR)AllocRaster(WIDTH,
          HEIGHT);
if (bitmxy->Planes[i] == NULL)
{
printf("xy bitmap planes were null\n");
goto quit; }
BltClear(bitmxy->Planes[i],planesize,1);
}
newscrxy.CustomBitMap = bitmxy;
scrxy = (struct Screen *)
          OpenScreen(&newscrxy);
if (scrxy==NULL)
{
printf("Unable to open screen xy.\n");
goto quit;
 }
rpxy = &scrxy->RastPort;
vpxy = &scrxy->ViewPort;
LoadRGB4(vpxy,cmap,16);
nw.Screen = scrxy;
w = OpenWindow(&nw);
if (w==NULL)   goto quit;
bitmzy = (struct BitMap *)
AllocMem(sizeof(struct BitMap),MEMF_CHIP);
if (bitmzy==NULL)
{
printf("No memory for zy bmap struct. \n");
goto quit;
}
InitBitMap(bitmzy,DEPTH,WIDTH,HEIGHT);
for (i = 0; i < DEPTH; i++)
{
          bitmzy->Planes[i] =
          (PLANEPTR)AllocRaster(WIDTH,HEIGHT);
          if (bitmzy->Planes[i] == NULL)
          {
          printf("yz bitmap planes were null\n");
          goto quit;
          }
BltClear(bitmzy->Planes[i],planesize,1);
}
newscrzy.CustomBitMap = bitmzy;
scrzy = (struct Screen *)
          OpenScreen(&newscrzy);
if (scrzy==NULL)
{
printf("Unable to open screen zy.\n");
goto quit;
}
          rpzy = &scrzy->RastPort;
          vpzy = &scrzy->ViewPort;
          LoadRGB4(vpzy,cmap,16);
          bitmxz = (struct BitMap *)
          AllocMem(sizeof(struct BitMap),MEMF_CHIP);
          if (bitmxz == NULL)
          {
          printf("No memory for xz bitmap struct.\n");
          goto quit;
          }
          InitBitMap(bitmxz,DEPTH,WIDTH,HEIGHT);
          for (i=0; i < DEPTH; i++)
          {
          bitmxz->Planes[i] =
                    (PLANEPTR)AllocRaster(WIDTH,HEIGHT);
          if (bitmxz->Planes[i] == NULL)
          {
          printf("No memory for xz bitmap planes.\n");
          goto quit;
          }
          BltClear(bitmxz->Planes[i],planesize,1);
          }
          newscrxz.CustomBitMap = bitmxz;
          scrxz = (struct Screen *)
                    OpenScreen(&newscrxz);
          if (scrxz==NULL)
          {
          printf("Couldn't open screen xz.\n");
          goto quit;
          }
          rpxz = &scrxz->RastPort;
          vpxz = &scrxz->ViewPort;
          LoadRGB4(vpxz,cmap,16);
          x1 = 0.0;
          y1 = 0.0;
          z1 = 0.0;
          count=0;
          count2=0;
          number=1;
          itoa(number,str);
          strcpy(outfile,"ram:");
          strcat(outfile,argv[1]);
          strcat(outfile,"set.");
          strcat(outfile,str);
          op = fopen(outfile,"w");
          tp = fopen("ram:points","w");
forever:;
          while((mess=(struct IntuiMessage *)
                    GetMsg(w->UserPort)) != NULL)
          {
                    class = mess->Class;
                    code = mess->Code;
                    ReplyMsg(mess);
                    if (class == CLOSEWINDOW)   goto finish;
                    if ((class== VANILLAKEY)&&(code=='q'))
                    goto quit;
          }
          /* Pick a set of specifications at random. */
                    r = rand() % ntrans;
                    cosza = cos(za[r]);
                    sinza = sin(za[r]);
                    cosxa = cos(xa[r]);
                    sinxa = sin(xa[r]);
                    cosya = cos(ya[r]);
                    sinya = sin(ya[r]);
                    xo = xf[r] + xt[r];
                    yo = yf[r] + yt[r];
                    zo = zf[r] + zt[r];
                    cxf =xf[r]*(1-xs[r]);
                    cyf =yf[r]*(1-ys[r]);
                    czf =zf[r]*(1-zs[r]);
                    x = x1;
                    y = y1;
                    z = z1;
/* first scale */
          if((xs[r])||(ys[r])||(zs[r]))
          {
```

```
                xxs = x*xs[r]  + cxf;
                yys = y*ys[r]  + cyf;
                zzs = z*zs[r]  + czf;
                x = xxs;
                y = yys;
                z = zzs;
        }
/* next translate */
        if((xt[r])||(yt[r])||(zt[r]))
        {
                xxt = x + xt[r];
                yyt = y + yt[r];
                zzt = z + zt[r];
                x = xxt;
                y = yyt;
                z = zzt;
        }
/* then rotate */
        if(ya)
        {
                zr = zo + (z - zo)*cosya + (x-xo)*sinya;
                xr = xo + (x - xo)*cosya - (z-zo)*sinya;
                x = xr;
                z = zr;
        }
        if(za)
        {
                xr = xo + (x - xo)*cosza - (y-yo)*sinza;
                yr = yo + (y - yo)*cosza + (x-xo)*sinza;
                x = xr;
                y = yr;
        }
        if(xa)
        {
                zr = zo + (z - zo)*cosxa - (y-yo)*sinxa;
                yr = yo + (y - yo)*cosxa + (z-zo)*sinxa;
                y = yr;
                z = zr;
        }
        x2 = x;
        y2 = y;
        z2 = z;
        nextx = (LONG)(xscale*x2 + xoffset);
        nexty = (LONG)(yscale*y2 + yoffset);
        nextz = (LONG)(yscale*z2 + zoffset);
        x1 = x2;
        y1 = y2;
        z1 = z2;
        cc2 = (r  % 14)  +  1;
        SetAPen(rpxy,cc2);
        SetAPen(rpzy,cc2);
        SetAPen(rpxz,cc2);

if((nextx>0)&&(nextx<640)&&(nexty>0)&&(nexty<400))
        {
                WritePixel(rpxy,nextx,nexty);
        }

if((nextz>0)&&(nextz<640)&&(nexty>0)&&(nexty<400))
        {
                WritePixel(rpzy,nextz,nexty);
        }

if((nextx>0)&&(nextx<640)&&(nextz>0)&&(nextz<400))
        {
                nextz = 400 - nextz;
                WritePixel(rpxz,nextx,nextz);
        }
        nextx = (LONG)x2;
        nexty = (LONG)y2;
        nextz = (LONG)z2;
        fprintf(op," %d %d ",cc2,nextx);
        fprintf(op," %d %d \n",nexty,nextz);
        count2++;
        /* When output file gets large close it and
        open another. */
        if(count2 > 12000)
        {
fprintf(op," %d  %d  %d  %d \n",dm,dm,dm,dm);
                fclose(op);
                count2=0;
                number++;
                itoa(number,str);
                strcpy(outfile,"ram:");
                strcat(outfile,argv[1]);
                strcat(outfile,"set.");
                strcat(outfile,str);
                op = fopen(outfile,"w");
        }
        /* Save the first 50 points. If they do not
        plot on the Amiga screen we can see where they
        are going and adjust scale and offsets. */
        count++;
        if(count < 50)
        fprintf(tp," %d %d ",r,nextx);
        fprintf(tp," %d %d \n",nexty,nextz);
        x2 = 0.0;
        y2 = 0.0;
        z2 = 0.0;
        nextx = 0;
        nexty = 0;
        nextz = 0;
        goto forever;
finish:;
        /* Save the three screens to ram:  Do not
        forget to convert these to ILBM and copy them
        to a disk if you wish to keep them. */
        strcpy(outfile,"ram:");
        strcat(outfile,argv[1]);
        strcat(outfile,"xy");
        ofhxy = Open(outfile,MODE_NEWFILE);
        if (ofhxy != NULL)
        {
        for ( i=0; i < DEPTH; i++)
        {
        nb = Write(ofhxy,bitmxy->Planes[i],planesize);
        }
        cols = &cmap[0];
        nb = Write(ofhxy,cols,colorsize);
        Close(ofhxy);
        }
        strcpy(outfile,"ram:");
        strcat(outfile,argv[1]);
        strcat(outfile,"zy");
        ofhzy = Open(outfile,MODE_NEWFILE);
        if (ofhzy != NULL)
        {
        for ( i = 0; i < DEPTH; i++)
        {
        nb = Write(ofhzy,bitmzy->Planes[i],planesize);
        }
        cols = &cmap[0];
        nb = Write(ofhzy,cols,colorsize);
        Close(ofhzy);
        }
        strcpy(outfile,"ram:");
        strcat(outfile,argv[1]);
        strcat(outfile,"xz");
        ofhxz = Open(outfile,MODE_NEWFILE);
        if (ofhxz != NULL)
        {
        for ( i = 0; i < DEPTH; i++)
        {
        nb = Write(ofhxz,bitmxz->Planes[i],planesize);
        }
        cols = &cmap[0];
        nb = Write(ofhxz,cols,colorsize);
        Close(ofhxz);
        }
quit:;
        if(op)
        {
        fprintf(op," %d  %d  %d  %d \n",dm,dm,dm,dm);
        fclose(op);
```

```
        }
        if(fp)  fclose(fp);
        if (w)  CloseWindow(w);
/* Close the screens and free the super bitmaps */
        if (scrxy) CloseScreen(scrxy);
        for (i = 0; i < DEPTH; i++)
        {
                if (bitmapy->Planes[i] != 0)
                {
                FreeRaster(bitmapy->Planes[i],
                        WIDTH,HEIGHT);
                }
        }
        if (bitmapy)
        FreeMem(bitmapy,sizeof(struct BitMap));
        if (scryz) CloseScreen(scryz);
        for (i = 0; i < DEPTH; i++)
        {
                if (bitmapx->Planes[i] != 0)
                {
                FreeRaster(bitmapx->Planes[i],
                WIDTH,HEIGHT);
                }
        }
        if (bitmapx)
        FreeMem(bitmapx,sizeof(struct BitMap));
        if (scrxz) CloseScreen(scrxz);
        for (i = 0; i < DEPTH; i++)
        {
                if (bitmapz->Planes[i] != 0)
                {
                FreeRaster(bitmapz->Planes[i],
                WIDTH,HEIGHT);
                }
        }
        if (bitmapz)
        FreeMem(bitmapz,sizeof(struct BitMap));
        /* Add a record of this run in the directory
        log "Run_Notes". */
        fp =fopen("Run_Notes","a");
        if (fp != NULL)
        {
        fprintf(fp,"\n 3-D Histogram %s ",argv[0]);
        fprintf(fp," %s ",argv[1]);
        fprintf(fp," %2.2f %2.2f ",xscale,xoffset);
        fprintf(fp," %2.2f %2.2f ",yscale,yoffset);
        fprintf(fp," %2.2f %2.2f \n",zscale,
zoffset);
        for(i = 0; i < npoints ; i++)
        {
        fprintf(fp," %4.2f %4.2f
%4.2f",xx[i],yy[i],zz[i]);
        fprintf(fp," %2.2f %2.2f
%2.2f",tx[i],ty[i],tz[i]);
        fprintf(fp," %3.3f %3.3f %3.3f
\n",xm[i],ym[i],zm[i]);
        fprintf(fp," %4.0f %4.0f
%4.0f\n",xt[i],yt[i],zt[i]);
        }
        printf("\n   Input comment for Run_Notes
...<RETURN>ends...\n");
        scanf("%s",comment);
        fprintf(fp,"%s \n",comment);
        fclose(fp);
        }
        if (IntuitionBase)
        CloseLibrary(IntuitionBase);
        if (GfxBase)
        CloseLibrary(GfxBase);
}  /* end of main */
/* itoa is from Kernighan and Ritchie */
itoa(nt,str)
char str[];
int  nt;
{
        int c, i, j, sign;
        if ((sign = nt) < 0)
        nt = -nt;
        i = 0;
        do {
        str[i++] = nt % 10 + '0';
        } while ((nt /= 10) > 0);
        if (sign < 0)
        str[i++] = '-';
        str[i] = '\0';
        for (i = 0, j=strlen(str) -1; i<j; i++,j--)
        {
        c = str[i];
        str[i]=str[j];
        str[j] = c;
        }
        return(0);
}
```

Listing 2

```
/* LISTING 2.  - RoraView - Reads object points,
plots x-y plane, rotates object about lines
parallel with the x, y and z axis, saves a View.
        Copyright 1992 by Laura M. Morrison  */

#include<stdio.h>
#include "libraries/dosextens.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "exec/exec.h"
#include "math.h"
#define DEPTH       4L
#define WIDTH       640L
#define HEIGHT      400L
#define MAXPOINTS  5000L
USHORT map[] = { 0x000,0x100,0x0f0,0xc10,
                        0x810,0x0ff,0x11f,0xfff,
                        0xc00,0x0f0,0x00f,0xc0c,
                        0x0cc,0xcc0,0xccc,0x777 };
SHORT  *pxl;
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
extern struct Window *w;
extern struct Screen *scr;
extern struct BitMap *bim;
extern struct RastPort *rp;
extern struct ViewPort *vp;
struct NewScreen newscr = {
        0,0,640,400,4,0,1,
        HIRES|LACE,
        CUSTOMSCREEN,
        NULL,NULL,NULL,NULL   };
struct Screen *scr;
struct NewWindow nw = {
        0,0,640,400,0,1,
        INTUITICKS|CLOSEWINDOW|VANILLAKEY,
        WINDOWCLOSE|ACTIVATE|BORDERLESS|
        WINDOWDRAG|SUPER_BITMAP,
        NULL,NULL,NULL,NULL, NULL,
        0,0,100,80,
        CUSTOMSCREEN        };
struct Window *w;
struct RastPort *rp;
struct ViewPort *vp;
struct BitMap *bim;
struct FileHandle *Open();
char str[8];
char outfile[20];
char infile[20];
static int nt[MAXPOINTS];
static float xxx[MAXPOINTS];
static float yyy[MAXPOINTS];
```

```
static float [illegible](MAXPOINTS);
static int [illegible];
static int [illegible];
static int [illegible];
static float [illegible];
static float [illegible];
static float [illegible];
static [illegible];
main(argc, argv)
int argc;
char *argv[];
{
        ULONG class;
        struct IntuiMessage *mess;
        [illegible];
        [illegible];
        [illegible];
        int [illegible], i;
        int [illegible], number;
        float [illegible];
        float [illegible];
        float [illegible];
        FILE *fopen(), [illegible];
        [illegible];
        [illegible];
        if ( argc < 2 )
        {
        printf("Need name of object point set.\n");
        exit(10);
    }

        if ((pp = fopen("pvparams","r"))==NULL)
        {
        printf("Need a 'pvparams' file with\n");
        printf("[illegible]\n");
        printf("[illegible]");
        printf("[illegible]\n");
        exit(1);
        }
        fscanf(pp,"[illegible]", [illegible]);
        fscanf(pp,"%f",&[illegible]);
        fscanf(pp,"[illegible]", [illegible]);
        fscanf(pp,"%f",&[illegible]);
        fscanf(pp,"[illegible]", [illegible]);
        fscanf(pp,"[illegible]", [illegible]);
        fscanf(pp,"[illegible]", [illegible]);
        fclose(pp);
        IntuitionBase = (struct IntuitionBase *)
                OpenLibrary("intuition.library",0);
        if(IntuitionBase == NULL)
        {
        printf("Couldn't open intuition library\n");
        exit(1);
        }
  GfxBase = (struct GfxBase *)
                OpenLibrary("graphics.library",0);
        if(GfxBase == NULL)
        {
        printf("Couldn't open graphics library\n");
        goto finish;
        }
        printf("[illegible]");
        printf("                     ");
        printf(" VIEW, ROTATE, SAVE 3-D point-set");
        printf("[illegible]\n");
        printf("                     ");
        printf("        Copyright (c) 1992 by \n");
        printf("                     ");
        printf("                         ");
        printf("Laura K. Morrison[illegible]");
        printf("                    [illegible]");
        printf("                         ");
        printf("[illegible]");
        printf("                     ");
        printf("[illegible]");
        printf("                     ");
        printf("Key 'x' or 'y' or 'z'");
        printf(" to rotate\n[illegible]");
        printf("                     ");
        printf("Key [illegible] before 'x', 'y', or 'z' [illegible]");
        printf(" reverse rotate.\n[illegible]");
        printf("                     ");
        printf("Key 's' to save screen.");
        printf("Key 'q' to quit. \n\n");
        [illegible];
        fp = fopen(argv[1],"r");
        if( fp == NULL)
        {
        printf("Can't open object point set. \n");
        exit(1);
        }
        for( i = 0; i < MAXPOINTS; i++)
        {
        fscanf(fp," %d %d ",&xx,&xx);
        fscanf(fp," %d %d ",&yy,&zz);
        if ( [illegible] == 999 || i > MAXPOINTS - 2 )
        {
                fclose(fp);
                goto rotate;
        }
        [illegible];
        xxx[i] = (float)xx;
        yyy[i] = (float)yy;
        zzz[i] = (float)zz;
        npoints++;
        }
rotate:
        planesize = (LONG)(WIDTH/8)*HEIGHT;
        bitm = (struct BitMap *)
        AllocMem(sizeof(struct BitMap),MEMF_CHIP);
        if (bitm==NULL)
        {
        printf("No memory for bmap structure.\n");
        goto finish;
        }
        InitBitMap(bitm,DEPTH,WIDTH,HEIGHT);
        for (i=0; i < DEPTH; i++)
        {
        bitm->Planes[i]=(PLANEPTR)
                                AllocRaster(WIDTH,HEIGHT);
                if (bitm->Planes[i] == NULL)
                {
                        printf("bitmap planes were null\n"
                        goto finish;
                }
                BltClear(bitm->Planes[i],planesize,1);
        }
```

—The complete Source Code and executable files may be found on the AC's TECH Disk

Pleasewrite to:
Laura Morrison
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Linking with HotLinks

by Dan Weiss

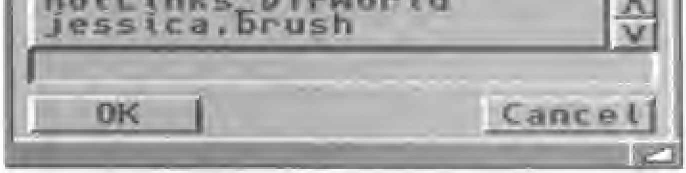
In the spring of this year a new third party *extension* to the Amiga OS was introduced. It offers a powerful inter-process data exchange and update mechanism. Its called *HotLinks*. The first products to use HotLinks were *PageStream*, the desktop publisher, and two programs that shipped with HotLinks, *BME* and *PageLiner*. BME is a bitmap editor and PageLiner is a formatted text processor. The next commercial programs to use HotLinks were *ProWrite* from New Horizons and *ImageMaster* from BlackBelt Systems. More commercial developers are implementing this standard, but many developers are unsure what HotLinks is, and how to use it. This article will help you both understand and include HotLinks in your programs.

What Is HotLinks?

HotLinks is the name for a system of programs, libraries, and data formats that allow the smooth exchange of data between programs on the Amiga. The HotLinks system is made up of:

HotLinks: The resident code
hotlinks.library: The shared library
several support programs

To make HotLinks complete requires one or more programs sharing data via HotLinks. HotLinks is *not* ARexx.



Why HotLinks?

There are many solutions to inter-process file sharing. The simplest and most well-known is the standard OS file system. Save the file in one program and open it in another. This is the universal system that ships with all Amigas. HotLinks use the standard OS as a model with two important extensions: updates and consistent file formats.

Updates in HotLinks enable any number of programs to keep up with the current status of any file that they are linked too. This means that as soon as a change is made in one program, the change can be reflected in several other programs without any user intervention. Consistent file formats is the logical application of IFF. The distinction is that not only do the formats meet the needs of all users, but all users use the *same* format. Proprietary formats are great for saving program-specific formats in program-specific manners, but a good interchange can be used by many programs. Furthermore, all HotLinks programs use only the agreed-on formats when publishing to HotLinks, assuring that each data type is passed in only one way. Notification and consistent file formats give HotLinks its Rjust worksS ease of use, and makes it usable by even the most novice of users.

How Does It Work?

Average HotLinks users place HotLinks in their WBStartup drawer. This way it is always ready to facilitate data-sharing. When a program wants to access the HotLinks system, it uses the shared library to message the resident program. The resident program also makes sure that when a file is updated, any subscribing program is notified, even if it is not running at the time. Through the shared library the program can create standard data structures for handling Publication (HotLinked files) information. Using the library calls to create and handle these data structures insures compatibility even if the structures change significantly in the future. The library also serves the important function of insulating the programmer from the Rdown and dirtyS details of how the file is exchanged. In this way massive changes and improvements can be made in HotLinks without breaking any programs. The important thing to remember is that HotLinks is designed not just for today, but for today and tomorrow.

Links In a Chain

So, one programmer to another, how does it work? Here is a call-level explanation of a HotLinks exchange. First let's publish a document. First we try to open the shared library; if the HotLinks resident program is not running ,the library won't open and we stop here. After we get the library open we use the first HotLinks call HLRegister to register ourselves with the HotLinks resident program.

HLRegister asks the calling program to supply a four -haracter ID for itself, a message port for notify messages, and a screen pointer so that

—What is HotLinks?
—Why HotLinks?
—How Does it Work?

HotLinks can open its requesters on the proper screen. HLRegister returns a handle that is used in subsequent calls to identify the calling program. This call, in effect, lets HotLinks know who you are and how to get your attention.

Next we need to create a Publication Block. A Publication Block, or pblock as it is known, is the standard data structure used for passing HotLinks-related information around. Instead of having to change your code every time there is a change in the pblock structure, we call AllocPBlock to create an empty pblock for us.

```
[illegible]
```

We pass our HotLinks handle and a receive a pointer to the pblock.

In this example we are looking to subscribe to an existing HotLinks Edition so we need to call GetPub to open the standard HotLinks edition requester.

```
[illegible]
```

The HotLinks edition requester looks like a standard file requester, but isn't since HotLinks deals only with its own files. GetPub needs the newly created pblock and a pointer to a simple filter procedure. Since there are several different kinds of HotLinks editions, a filter procedure is used so that the user is given a choice of only those files that would work with a given program. The filter procedure typically looks at the type in the editions pblock to determine if is an edition that should be listed in the requester. A good example of a simple filter proc is included in an example program in the HotLinks developer kit. When the user has made his choice, GetPub will return with the pblock full of information about the edition. At this point HotLinks and the standard file system begin to look alike.

If we want to modify the data in any way we must get a write lock. This prevents anyone from getting permission to modify the data while we may be modifying it. This is very important since the basic assumption of HotLinks is that more than one program will be accessing the same data at the same time. To get a write lock, we need to call LockPub.

```
[illegible]
```

LockPub takes the editions pblock and some flags that indicate if we want a read lock or write lock, or we want to unlock a locked file. It is important to note that HotLinks locks are RsoftS. This means that if the machine is rebooted, then all locks are lost. This is fine since no program should be able to permanently hold a lock on an edition.

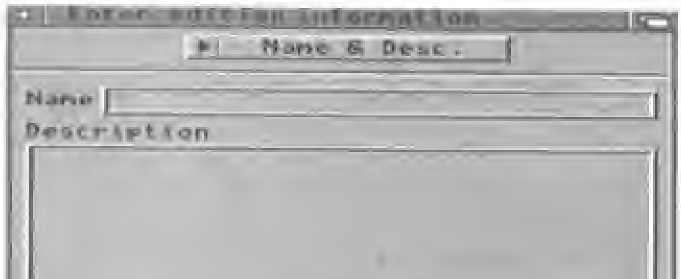
After obtaining a write lock, the pblock is passed to the OpenPub call to gain access to the edition.

```
int error = OpenPub(struct PubBlock *pblock, int
flags)
```

Now that you have both the file and a lock on it, you can access and modify it as you see fit using the ReadPub, WritePub, and SeekPub calls.

```
int error = ReadPub(struct PubBlock *pblock,char
*buffer, int len)
int error = WritePub(struct PubBlock *pblock,char
*buffer, int len)
int error = SeekPub(struct PubBlock *pblock, int
offset, int flags)
```

For the most part, these work like standard OS calls.



The trick now is what to read and write. HotLinks files are very similar to their IFF counterparts, usually with a few minor extensions. The bitmap format has specific extensions to the BMHD chunk defining what type of picture that data represents—Palette, Grayscale, RGB, RGBA,CMYK, CMYKA—and the resolution of the image. The text format, DTXT, is a completely new IFF format. The

reason for this is that it goes well beyond ASCII to provide complex text formatting that professional word processors need. But as you will see in the example programs, it can also be simple enough for basic text use. The really important thing to note here is that there is only one text format and only one bitmap format, not 10 of each. This means once you support HotLinks text files you never have to write another chunk of code to support text via HotLinks.

After an edition has been modified you will want to publish the changes. This is called updating. To update a file, simply write out a new version of the edition and close it with ClosePub.

```
int error = ClosePub(struct PubBlock *pblock);
```

This lets HotLinks know that this is a new version of the edition. All files that have notify set for this edition will be notified of the change in version.

Since you have accessed the file, you may also want to be notified of any further changes by other users. The Notify call is used to set up a link between HotLinks and the calling program concerning a given edition.

```
int error = Notify(struct PubBlock *pblock, int flag, int ..., void
*userdata);
```

Notify takes the pblock for the edition you want the notify message for, along with a flag that indicates if you are setting up the notify, canceling the notify, or setting up an exclusive notify. The notify message itself will be sent to the message port that was specified in the HLRegister call. The message will be an HLMsg struct.

Normally it is simplest to set up a notify on an edition when you first start using it and remove it when you are finally finished with the edition. Doing this means you will receive a notify message for the changes you make as well as those made by other programs. To prevent rereading an edition you have modified, use the PubStatus call to see if the current edition is different from the one you have.

```
int error = PubStatus(struct PubBlock *pblock);
```

PubStatus will return the status of the edition specified by the pblock, in the return error.

When you are finally done working with an edition, you need to cancel the notify and free the pblock. You cancel the notify with another call to Notify. To free the pblock, call FreePBlock.

```
int error = FreePBlock(struct PubBlock *pblock);
```

It is important use the FreePBlock call since the pblock may likely change size in the future. Remember each notify should be cleared and each pblock should be freed before finally leaving HotLinks.

To leave HotLinks, you must make one last call, UnRegister.

```
int error = UnRegister(int handle);
```

UnRegister frees the handle allocated by HLRegister and lets HotLinks know that the calling program is no longer connected with the HotLinks system.

When you are publishing a new edition to HotLinks, the sequence is much the same. First you register with HLRegister if you are not already registered, allocate a pblock with AllocPBlock and then call PutPub.

```
int error = PutPub(struct PubBlock *pblock, int(*filterproc)());
```

PutPub presents the standard HotLinks requester for publishing a new edition. It allows the user to name the edition as well as set certain security information. PutPub accepts the standard pblock and will display the Name, Description, and Access information if it is present. This is the proper way to suggest default values to the user. The filterproc parameter is not used at this time.

Once PutPub has filled in the pblock you can set up a standard notify with Notify, get a write lock with LockPub, and access the file as you did in the example above with OpenPub, WritePub, SeekPub and ClosePub. Be sure to always get a write lock when modifying an edition. Finally, when you are done, cancel the notify and free the pblock. When you are done with HotLinks, exit cleanly using the UnRegister call.

Other Calls

A simple look at the HotLinks documentation will show several calls that have not been discussed. These fall into two categories. The first is user access management. SetUser sets the current user of the HotLinks system. Generally this is not necessary as the users are already logged in. Use of this call is best left to the login process. NewPassword and ChgPassword also fall into this category, and again are best left to HotLinks support programs.

The other category of HotLinks could be called Rbehind the scenesS. For the most part you will not need to use these calls, but you can if you need to get a little deeper into HotLinks. RemovePub deletes an edition if the user has access to do so. This is used mostly as a cleanup measure when edition can not be properly completed. FirstPub gets the first edition that is available. Note that it does not use a filter proc so this may give you a pblock to any edition. NextPub, and successive calls to NextPub, will give you a pblock to the next edition in line. These two calls allow you to search the entire set of editions. GetInfo will fill in a pblock that has the i.d. of an edition set in it. This is how you can get a full pblock without forcing the user to reselect the edition via GetPub. It is suggested that when you save a file with a HotLinks edition in it, you store the i.d. and version number from the pblock as well as the actual data. Storing the i.d. and version number only forces you to use GetInfo and have an up-to-date pblock. SetInfo changes an editions pblock to match the supplied pblock. Be sure to call GetInfo first to fill the pblock so that information you do not change will be maintained. Finally PubInfo allows the user to update or change the information associated with an edition. The PubInfo requester is the same as the PutPub requester.

Real World Example

Now that we have run through a logical example, lets run through an actual program. The program is called text2hl. The complete code for this program along with code for several other HotLinks programs is included with the HotLinks developer kit.

```
/*
 *
 * text2hl.c - publish a standard ascii file to HotLinks
 *
 */
```

```
#include [illegible]
#include [illegible]
#include [illegible]
#include [illegible]
#include [illegible]
#include [illegible]
#include [illegible]

[illegible]
[illegible]

/* hotlinks library base pointer */
[illegible]

/* [illegible] publication block pointer */
[illegible]

/* hotlinks [illegible] */
[illegible]

/* version string */
[illegible]



/* forward declarations */
[illegible]
void shutdown();



int main(argc, argv)
int argc;
char *argv[];
{
        [illegible]
        [illegible]
        [illegible]
        [illegible]

        if(argc[illegible])
        {
                printf([illegible]);
                exit(0);
        }
```

The first thing the program does is to try to open the hotlinks.library. If HotLinks is not running, the library cannot be opened and you should give up right here.

```
        /* try to open the hotlinks library
         * The library will not open unless HotLinks is running
         */
        if((HotLinksBase=[illegible]
OpenLibrary("hotlinks.library", 0))==0)
        {
                printf("ERROR - could not open the
hotlinks.library\n");
                exit(0);
```

After opening the library, you need to register with HotLinks.

```
        /* register this program with the hotlinks system */
        [illegible]
```

TEST is passed as the signature for this example, and since it will not be receiving any msgs, NULL (0) is passed for the MsgPort parameter. The program is running on the workbench so NULL (0) is passed for the screen parameter.

AllocPBlock is used next to create a pblock that will be used to publish the new edition. It is important to check the pblock to make sure that is was allocated, and did not return an error. For now the HotLinks side is ready. We need to check the file size of, and open the text file that is to be published.

```
        /* get a [illegible] */
        [illegible]

        /* check for errors */
        [illegible]
[illegible])
        {
                printf("ERROR - AllocPBlock [illegible]
[illegible]");
                shutdown();
                exit(0);
        }

        /* get the file size */
        [illegible]
        if(!lock)
        {
                printf("ERROR - [illegible]
[illegible]");
                shutdown();
                exit(0);
        }
        if(!Examine(lock, [illegible]))
        {
                printf("ERROR - could not examine input file\n");
                UnLock(lock);
                shutdown();
                exit(0);
        }
        [illegible]
        UnLock(lock);

        /* open the input file */
        if(!(fh=Open(argv[1], MODE_OLDFILE)))
        {
                printf("ERROR - [illegible]
reading\n");
                shutdown();
                exit(0);
```

Once the source text file is open, stuff the pblock with Type, Access, and Name information, and call PutPub. PutPub presents the PutPub requester and allows the user to change any of the information. PutPub then updates the pblock so that all information is current. Again make sure that PutPub returns NOERROR before continuing. Open the publication for writing using the new pblock. If you are able to do this, it in effect gives you a write lock on the file, so no call to LockPub is needed. No actual lock is set, but HotLinks will not allow any program to gain a write lock when an edition is open for writing.

```
	/* set up some defaults */
	pb->PRec.Type = DTXT;
	pb->PRec.Access = ACC_DEFAULT;
	strcpy(pb->PRec.Name, argv[1]);

	/* get a publication using the publication requester
provided by the
	 * hotlink.library.
	 */
	error = PutPub(pb, 0);

	/* if the user selected a file and pressed ok then delete
the file*/
	if(error!=NOERROR)
	{
		/* check for errors */
		switch(error)
		{
				case NOPRIV: printf("ERROR: privilege
violation\n");
					break;

				case INVPARAM: printf("ERROR: invalid
parameters\n");
					break;
		}
		Close(fi);
		shutdown();
		exit(0);
	}
```

At this point we are simply writing out a normal IFF file to HotLinks using WritePub call. If for some reason the process fails before the edition is fully created, it is important to delete the edition with RemovePub, as it is an invalid edition.

```
	/* open the publication file */
	if((error=OpenPub(pb, OPEN_WRITE))!=NOERROR)
	{
		printf("ERROR - could not open the publication
for writing\n");
		Close(fi);
		shutdown();
		exit(0);
	}
```

```
	chunktype = FORM;
	WritePub(pb, (char *)&chunktype, 4); /* write the FORM tag
*/
	chunksize = filesize+12;
	WritePub(pb, (char *)&chunksize, 4); /* write the FORM size
*/
	chunktype = DTXT;
	WritePub(pb, (char *)&chunktype, 4); /* write the FORM type
(DTXT) */

	WritePub(pb, (char *)&chunktype, 4); /* write out the chunk
header (DTXT) */
	chunksize = filesize;
	WritePub(pb, (char *)&chunksize, 4);

	/* allocate memory to hold the chunk */
	if((buff=AllocMem(filesize, 0))==0)
	{
		printf("ERROR : out of memory\n");
		ClosePub(pb);
		Close(fi);
		RemovePub(pb);
		shutdown();
		exit(0);
	}

	/* read the chunk data in */
	Read(fi, buff, filesize);

	newlen = writetext(pb, buff, filesize);

	/* pad the file out to an even byte */
	if(filesize&0x00000001)
	{
		*buff = 0;
		WritePub(pb, buff, 1);
	}

	/* free the memory used */
	FreeMem(buff, filesize);

	/* close the input file */
	Close(fi);

	/* adjust the chunk lengths for the commands written out */
	SeekPub(pb, 4, SEEK_BEGINNING);
	chunksize = newlen+12;
	WritePub(pb, (char *)&chunksize, 4);
	SeekPub(pb, 8, SEEK_CURRENT);
	chunksize = newlen;
	WritePub(pb, (char *)&chunksize, 4);
	SeekPub(pb, 0, SEEK_END);
```

Since the edition is in DTXT format, we need to use the routine writetext to publish the DTXT version of the source text file. As you can see, this is a pretty straightforward process since only the tab and newline characters need to be translated for the DTXT file. The translation takes the form of creating DTXT tags for each command to replace the single character code. Writetext returns the

actual length of the DTXT data written since tabs and newline tags will cause the data to expand.

```
int writetext(pb, buff, len)
struct PubBlock *pb;
char *buff;
int len;
{
        int flen;
        char command, cflag, clen;

        flen = 0;
        while(len)
        {

                switch(*buff)
                {
                        /* write out the TEXT_TAB command */
                        case 0x09: command = TEXT_TAB;
                                cflag = TEXT_FLAGS_TAB;
                                clen = 0;
                                WritePub(pb, &clen, 1);
                                WritePub(pb, &command, 1);
                                WritePub(pb, &cflag, 1);
                                WritePub(pb, &clen, 1);
                                flen+=4;
                                break;

                        /* write out the TEXT_NEWLINE command */
                        case 0x0a: command = TEXT_NEWLINE;
                                cflag = TEXT_FLAGS_NEWLINE;
                                clen = 0;
                                WritePub(pb, &clen, 1);
                                WritePub(pb, &command, 1);
                                WritePub(pb, &cflag, 1);
                                WritePub(pb, &clen, 1);
                                flen+=4;
                                break;

                        /* write out the text itself */
                        default: WritePub(pb, buff, 1);
                                flen++;
                                break;
                }
                len--;
                buff++;
        }
        return(flen);
}
```

When the edition is finished, or if the program aborts, there is still some cleanup that must take place. This is contained in the shutdown procedure. First free the pblock with FreePBlock, then unlink from HotLinks via UnRegister. Finally close the library with the standard DOS call.

```
        /* close the publication */
        ClosePub(pb);
        shutdown();
}


void shutdown()
{
        if(pb)
        {
                /* free the publication block pointer */
                FreePBlock(pb);
        }
        if(hlh)
        {
                /* unregister this program from HotLinks */
                UnRegister(hlh);
        }
        if(HotLinksBase)
        {
                /* close the library */
                CloseLibrary((struct Library *)HotLinksBase);
        }
}
```

Final Notes

There you have it. Not too hard is it? While I was writing this article, I was struck with just how easy it is to implement HotLinks. The documentation can be a little scary, but in practice it is no different than reading and writing normal files. If your program already reads or writes IFF ILBM files, then you are inches away from exchanging files with PageStream, BME, and ImageMaster. Even better, the more programs that support HotLinks, the more programs you'll work with. HotLinks is pretty exciting, and it will be getting even more exciting in the future. Remember, though, that only a program that follows the rules gets to keep up with the fun. Be sure to get write locks!

✓

Please write to:
Dan Weiss
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

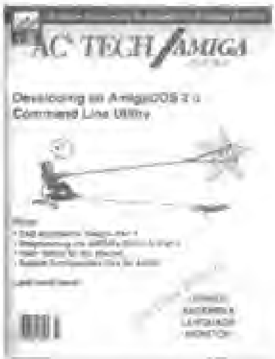
AC's TECH

AC's TECH Volume 1 Number 1

Adapting Mattel's Power Glove to the Amiga by Paul King and Mike Cargal
AmigaDOS for Programmers by Bruno Costa
AmigaDOS, EDIT and Recrusive Programming Techniques by Mark Pardue
An introduction to Interprocess Communication with ARexx by Dan Sugalski
An Introduction to the ilbm.library by Jim Fiore
Building the VidCell 256 Grayscale Digitizer by Todd Elliott
Creating a Database in C Using dBC III by Robert Broughton
FastBoot: A Super BootBlock by Dan Babcock
Magic Macros with ReSource by Jeff Lavin
Silent Binary Rhapsodies by Robert Tiess
Using Intuition's Proportional Gadgets from FORTRAN 77 by Joseph R Pasek

AC's TECH Volume 1 Number 2

A Mega and a Half on a Budget by Bob Blick
Accessing Amiga Intuition Gadgets from a FORTRAN Program Part II-Using Boolean Gagdets by Joseph R Pasek
Adding Help to Applications Easily by Philip S Kasten
CAD Application Design: Part I - World and View Transforms by Forest W Arnold
Interfacing Assembly Language Applications to ARexx by Jeff Glatt
Intuition and Graphics in ARexx Scripts by Jeff Glatt
Programming the Amiga's GUI in C Part I by Paul Castonguay
ToolBox Part I: An Introduction to 3D Programming by Patrick Quaid
UNIX and the Amiga by Mike Hubbart



AC's TECH Volume 1 Number 3

Accessing the Math Co-Processor from BASIC by R P Haviland
C Macros for ARexx by David Blackwell
CAD Application Design Part II by Forest W Arnold
Configuration Tips for SAS-C by Paul Castonguay
Hash for the Masses: An Introduction to Hash Tables by Peter Dill
Programming for HAM-E by Ben Williams
Programming the Amiga's GUI in C-Part II by Paul Catonguay
The Development of an AmigaDOS 2.0 Command Line Utility by Bruno Costa
Using RawDoFmt in Assembly by Jeff Lavin
VBRMon: Assembly Language Monitor by Dan Babcock
WildStar: Discovering An AmigaDOS 2.0 Hidden Feature by Bruno Costa

AC's TECH Volume 1 Number 4

GPIO Low Cost Sequence Control by Ken Hall
Language Extensions Strings of Type StringS by Jimmy Hammonds
Programming the Amiga's GUI in C Part III by Paul Castonguay
Programming with the ARexxDB Records Manager by Benton Jackson
State of Amiga Development Denver DevCon Address by Jeff Scherb
STOX An ARexx Based System for Maintaining Stock Prices by Jack Fox
The Development of a Ray Tracer Part 1 by Bruno Costa
The Varafire Solution Build Your Own Variable Rapid Fire Joystick by Lee Brewer
Using Interrupts for Animating Pointers by Jeff Lavin

Back Issue Index

AC's TECH Volume 2 Number 1

AudioPProbe-Experimets in Synthesized Sound with Modula 2 by Jim Olinger
CAD Application Design Part III by Forest W Arnold
Implementing an ARexx Interface in Your C Program by David Blackwell
Low-Level Disk Access in Assembly by Dan Babcock
Programming a Ray Tracer in C Part II by Bruno Costa
Programming the Amiga in 680x0 Assembler Part I by William P Nee
Programming the Amiga's GUI in C Part IV by Paul Castonguay
Spartan-Build Your Own SCSI Interface for Your Amiga 5000/1000 by Paul Harker
The Amiga and the MIDI Hardware Specification by James Cook
Writing Protocols for MusicX by Daniel Barrett

AC's TECH Volume 2 Number 2

Amiga Voice Recognition by Richard Horne
Animated Busy Pointer by Jerry Trantow
Blit Your Lines by Thomas Eshelman
Copper Programming by Bob D'Asto
Dynamically Allocated Arrays by Charles Rankin
Implementing an ARexx Interface in Your C/Program Part 2 by David Blackwell
Iterated Function Systems for Amiga Computer Graphics by Laura Morrison
Keyboard I/O from Amiga Windows by John Baez
MenuScript by David Ossorio
Programming the Amiga in Assembly Language Part 2 by William P Nee

AC's TECH Volume 2 Number 3

Backup_DOC by Werther Pirani
CAD Application Design Part 4 by Forest W Arnold
HighSpeed Pascal by David Czaya
PCX Graphics by Gary L Fait
Programming the Amiga in Assembly Language Part 3 by William P Nee
Programming the Amiga's GUI in C Part 5 by Paul Castonguay
Understanding the Console Device by David Blackwell

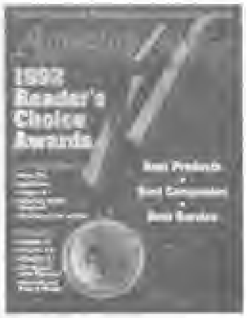
AC's TECH Volume 2 Number 4

Advanced Scripting by Douglas Thain
Entropy in Coding Theory by Joseph Graf
Fast Plots by Michael Griebling
Getconfirm() by John Baez
In Search of the Lost Windows by Phil Burke
No Mousing Around by Jeff Dickson
Programming the Amiga in Assembly Language part 5 by William P Nee
Putting the Input Device to Work by David Blackwell
Quarterback 5.0 a Review by Merrill Callaway
Tape Drives by Paul Gittings
The Joy of Sets by Jim Olinger
True BASIC Extensions by Paul Castonguay

Should You?

Amaze Them Every Month!

Amazing Computing For The Commodore Amiga is dedicated to Amiga users who want to do more with their Amigas. From Amiga beginners to advanced Amiga hardware hackers, AC consistently offers articles, reviews, hints, and insights into the expanding capabilities of the Amiga. *Amazing Computing* is always in touch with the latest new products and new achievements for the Commodore Amiga. Whether it is an interest in Video production, programming, business, productivity, or just great games, AC presents the finest the Amiga has to offer. For exciting Amiga information in a clear and informative style, there is no better value than *Amazing Computing*.



A Guide For Every Amiga User.

Give the Amiga user on your gift list even more information with a **SuperSub** containing *Amazing Computing* and the world famous ***AC's GUIDE To The Commodore Amiga***. *AC's GUIDE* (published twice each year) is a complete listing of every piece of hardware and software available for the Amiga. This vast reference to the Commodore Amiga is divided and cross referenced to provide accurate and immediate information on every product for the Amiga. Aside from the thousands of hardware and software products available, *AC's GUIDE* also contains a thorough list and index to the complete Fred Fish Collection as well as hundreds of other freely redistributable software programs. No Amiga library should be without the latest AC's GUIDE.

More TECH!

AC's TECH For The Commodore Amiga is an Amiga users ultimate technical magazine. *AC's TECH* carries programming and hardware techniques too large or involved to fit in *Amazing Computing*. Each quarterly issue comes complete with a companion disk and is a must for Amiga users who are seriously involved in understanding how the Amiga works. With hardware projects such as creating your own grey scale digitizer and software tutorials such as producing a ray tracing program, *AC's TECH* is the publication for readers to harness their Amiga to fulfill their dreams.

YES!

To order phone 1-800-345-3360

(in the U.S. or Canada)

Foreign orders: 1-508-678-4200

or

FAX 1-508-675-6002.

or

CLIP THIS COUPON AND MAIL IT TODAY!

MAIL TO:
Amazing Computing
P.O. Box 2140
Fall River, MA 02722-2140

YES! *The "Amazing" AC publications give me **3 GREAT** reasons to save! Please begin the subscription(s) indicated below immediately!*

Name ____________ DISCOVER

Address ____________

City ____________ State ______ ZIP ____________ MasterCard

Charge my: ☐ Visa ☐ MC # ____________

Expiration Date ________ Signature ____________ VISA

Please circle to indicate this is a **New Subscription** or a **Renewal**

1 year of **AC**	**12 big issues of Amazing Computing! Save over 49% off the cover price!**	**US $27.00** ☐ **Canada/Mexico $34.00** ☐ **Foreign Surface $44.00** ☐
1-year SuperSub	**AC + AC's GUIDE – 14 issues total! Save more than 46% off the cover prices!**	**US $37.00** ☐ **Canada/Mexico $54.00** ☐ **Foreign Surface $64.00** ☐
1 year of **AC's TECH**	**4 big issues! The ONLY Amiga technical magazine!**	**US $43.95** ☐ **Canada/Mexico $47.95** ☐ **Foreign Surface $51.95** ☐

Please call for all other Canada/Mexico/foreign surface & Air Mail rates.
Check or money order payments must be in US funds drawn on a US bank; subject to *applicable* sales tax.

MorphPlus™

Finally! True Cinematic Quality Morphing For The Amiga®!

ASDG is not the first to advertise "cinematic quality morphing" for the Commodore Amiga®. Having seen the other products, there's obviously more than one way to define that term.

To us, "cinematic quality morphing" means these things:

• Morphing must be fast.
In a production environment, time is money. **ASDG's MorphPlus™** is the fastest morphing product available for the Commodore Amiga®. **MorphPlus™** powers through complicated full overscan morphs 3 to 11 times faster than the other products.

Fastest ....MorphPlus™
Easiest-To-Use ....MorphPlus™
Highest Quality.......MorphPlus™

• Morphing must be easy.
Experts in the field praise the intuitive design of the **MorphPlus™** user interface which lets them create sophisticated full motion morphs in minutes instead of weeks.

• Morphing must be high quality
(so that it truly *can* be used for cinematic or professional video applications). **MorphPlus™** is already in use in Hollywood productions, replacing high end systems.

This is what we mean by "cinematic quality morphing."

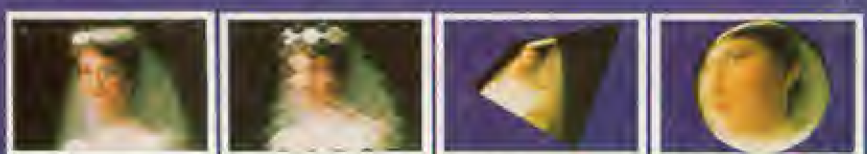
If these are the criteria *you* would use, then **MorphPlus™** is the choice you should make.

See it at your local dealer!

ASDG

925 Stewart Street Madison, WI 53713
608/273-6585

What's The Plus?



And More!